Temporal Planning for Rich Numeric Contexts

Bajada, Josef

Awarding institution:
King's College London

Temporal Planning for Rich Numeric Contexts

by

Josef Bajada

A thesis submitted in partial fulfilment

of the requirements for the degree of

Doctor of Philosophy

Department of Informatics

School of Natural and Mathematical Sciences

King's College, University of London

United Kingdom

2016

Abstract

Real-world planning problems often feature complex temporal and numeric characteristics. These include concurrent activities and also effects that involve continuous change. This work presents the formalism behind reasoning with required concurrency that involves continuous change in temporal planning problems, together with a set of techniques to solve a class of tasks that to date are hard to solve with current state-of-the-art temporal planners.

The motivation for this work is scenarios where planning actions have rich numeric effects on some physical system. One such example is automated demand dispatch for electricity provision, where actions that fulfil customer requirements have an effect on various metrics, such as wattage or costs, which could be subject to operational or commercial constraints.

An algorithm that handles discrete interference of linear continuous effects, referred to as constants in context, is presented. This technique allows discrete actions to update the rate of change of a continuous effect taking place concurrently. This work builds on techniques used in current temporal planners that make use of linear programming, and also extends the heuristic to guide the search to a solution. This algorithm was implemented in a new temporal and numeric planner called DICE and evaluated with some benchmark domains.

PDDL, the current de facto standard language for planning domains and corresponding planning tasks, was extended to support interactions with external class modules. The proposed extension, PDDLx, defines a generic planner-solver interface for both discrete and continuous effects. This enables planners that implement this interface to interact with external solvers and incorporate context-specific effects in a black-box fashion, enabling complex numeric behaviour to be encapsulated within such modules.

Non-linear monotonic continuous effects, defined in the proposed PDDLx extension, are integrated within the planner using a non-linear iterative convergence algorithm. It searches for a linear approximation within an acceptable configurable error margin, which is then used within the linear program computed for each temporal state. This algorithm proves to be effective in various domains where non-linear continuous behaviour is prevalent. This technique was implemented as an extension to DICE, called uNICOrn, which performs non-linear iterative convergence for continuous effects whose duration needs to be determined by the planner. uNICOrn was also evaluated with some benchmark non-linear domains.

A case study on the automated demand dispatch domain is presented to demonstrate the use of the planning algorithms proposed in this thesis. Linear and non-linear planning problems are evaluated and the performance of uNICOrn on these problem instances was analysed.

This work builds on current techniques used for temporal planning with continuous numeric behaviour using linear programming, and enhances them to remove some of their intrinsic limitations. The result is a set of algorithms that are more effective in solving real-world applications that involve continuous change and rich numeric behaviour.

Acknowledgements

This work would not have been possible without the academic, moral and financial support of various people and organisations.

I am deeply grateful to my supervisors, Professor Maria Fox and Professor Derek Long, for their guidance, inspiration and continuous encouragement. Apart from their in-depth knowledge, they provided me with unique opportunities to get international exposure within the A.I. community, which has enriched my experience even further.

This research was funded and supported by the UK Engineering and Physical Sciences Research Council (EPSRC) as part of the project entitled *The Autonomic Power System* (Grant Ref: EP/I031650/1). I would like to thank all the researchers participating in the project for their insight and interesting discussions during the various meetings and events.

I would also like to thank my colleagues and friends at King's College London, whose company and friendship made the whole experience even more pleasant and enjoyable. The regular meetings of the King's A.I. Planning Research Group were also instrumental to generate ideas and gather invaluable feedback throughout my research.

Last and foremost, I would like to show my utmost gratitude to my wife Thérèse, my parents, and the rest of our respective families for their immeasurable patience and support throughout my academic and professional endeavours.

Contents

List of Figures

List of Tables

Code Listings

List of Publications

During the course of this research, the following articles were peer reviewed and published:

- Josef Bajada, Maria Fox, and Derek Long (2013). "Load Modelling and Simulation of Household Electricity Consumption for the Evaluation of Demand-Side Management Strategies". In: *4th IEEE PES Innovative Smart Grid Technologies Europe (ISGT Europe). IEEE*

- Josef Bajada, Maria Fox, and Derek Long (2014a). "Challenges in Temporal Planning for Aggregate Load Management of Household Electricity Demand". In: *31st Workshop of the UK Planning & Scheduling Special Interest Group (PlanSIG)*

- Josef Bajada, Maria Fox, and Derek Long (2014b). "Temporal Plan Quality Improvement and Repair using Local Search". In: *Proceedings of the 7th European Starting A.I. Researcher Symposium (STAIRS-2014)*. IOS Press, pp. 41–50

- Josef Bajada, Maria Fox, and Derek Long (2015). "Temporal Planning with Semantic Attachment of Non-Linear Monotonic Continuous Behaviours". In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI-15)*, pp. 1523–1529

- Josef Bajada, Maria Fox, and Derek Long (2016). "Temporal Planning with Constants in Context". In: *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*

1. Introduction

Automated planning has a huge potential for use in a wide range of applications. While traditionally it was seen in the context of Artificial Intelligence (A.I.) for robotics and industrial automation, new technological advancements in various other domains have created new opportunities for the application of automated planning. Highly connected systems, such as telecommunications networks, transport and fleet management, and power grids are becoming too complex for people to manage manually in an efficient and effective way. These domains have the potential of gaining significant benefits from A.I. planning technology. It could be used to automatically find the right sequence of actions that achieve a certain goal, while satisfying technical and operational constraints, minimising costs or maximising some utility.

A.I. planning has seen significant advancement over the past couple of decades. While it is known to be a very hard problem from an algorithmic complexity point of view (Bylander, 1994; Rintanen, 2007), several approximation techniques and heuristic based methods have been proposed to improve performance. Automated planning typically involves modelling the domain, in terms of its possible facts and actions, modelling the problem instance that needs to be solved, with the context of the domain, and performing the search process to find a valid plan to achieve the goal.

Planning problems come in various flavours. *Classical planning* traditionally limits the scope to fully observable, deterministic, static environments with a single executive agent (Russell and Norvig, 2009) and propositional state variables, known as STRIPS (Fikes and Nilsson, 1971). However, real-world problems often involve other characteristics such as time, numbers, concurrency, partial observability, stochastic effects, exogenous interference, dynamic environments and multiple executive agents, potentially including adversarial behaviour.

The aim of this research is to enhance current planning technology to support real-world problems that feature rich temporal and numeric characteristics. The boundaries of current temporal and numeric planning techniques are pushed such that more complex numeric features can be incorporated, widening the class of problems supported by such technology. This involves extending both the modelling aspect and also the search process, and is proposed as an incremental enhancement to current state-of-the-art temporal and numeric planning technology, addressing some of the limitations found in these techniques.

1.1 Motivation and Scope

The motivation of this work comes from the need to model and plan for temporal planning domains with rich numeric characteristics. While the techniques developed for this thesis are

generic and domain-independent, they stem from a very specific domain. This is the problem of demand-side management of electricity load within a smart grid power network.

Currently, operational plans are drafted based on historical data and predictions based on weather, sunrise/sunset times and the type of day, including events and celebrations that might increase power consumption. This process does not take into account the complexities of renewable energy or flexible load, and often just involves a significant amount of power reserve being generated to be able to handle the occasional spikes in demand that might occur. Generators typically involve a long ramp up time, which means that they have to run even if they are not actually needed.

Smart grids are electricity networks that feature intelligent control points and sensors throughout their topology. These networks provide two-way communication from consumers, via the smart meters, to the suppliers. This communication infrastructure enables intelligent metering, monitoring, and remote actuation to control demand and supply. These capabilities are key to manage the power network efficiently, especially with an increasing proliferation of renewable energy and demand-side generation (such as solar panels on rooftops), which depend on the weather conditions irrespective of the demand, risking network destabilisation.

Demand-Side Management (DSM) (Strbac, 2008) is a set of techniques designed to influence the demand-side of the power network, such that it operates more efficiently and preserves grid stability. This includes indirect incentive based techniques, such as cheaper tariffs during off-peak times, and also more direct control of flexible demand, such as utility-controlled electric vehicle charging, water heating or dish washing. The primary objectives of these techniques are to shift electricity load such that peaks are reduced, and to increase the utilisation of cheaper and cleaner energy sources.

Automated *demand dispatch* (Brooks et al., 2010) is a DSM technique that involves deciding at which point, and in what way, various elements in the demand-side of the power grid should be activated, to satisfy consumer demand. An *aggregator* would record the information pertaining to the users' requests through the smart grid infrastructure. It would then plan ahead how these requests are going to be fulfilled, within the temporal constraints of the users and the operational constraints of the network. With a high proliferation of renewable generation, this plan could take into account expected supply levels for the predicted weather conditions and inflexible demand. It could also take into consideration the varying costs in the wholesale energy market, which is typically managed through auctions, and try to minimise the costs needed to satisfy the demand. The role of an aggregator is primarily to balance supply and demand in order to guarantee the stability of the network. However, it also has a commercial interest to optimise costs wherever possible.

The elements that can be controlled by the aggregator consist of user appliances and electricity storage devices. These loads should not affect the consumer's experience when controlled remotely. Examples of these types of loads include dishwashers, clothes washing machines and dryers, electric hot water boilers, and plug-in electric or hybrid vehicles. These activities are typically associated with temporal constraints, such as deadlines or operational time windows, specified by the user. Some of them also involve various activity profiles or modes. For example, an electric vehicle can be charged in slow (low wattage) mode, fast (high wattage)

mode, or even smart charged (intermittent charging controlled by the aggregator to interleave the load with other activities). Electricity storage devices can consist of either consumer owned equipment or assets installed and maintained by the aggregator. This equipment can harness surplus energy produced at a cheaper price so that it is then utilised during time periods that have a higher demand or a more expensive wholesale energy price.

This domain features various interesting temporal and numeric characteristics. Activities have temporal constraints on their durations, mainly operation times and deadlines by when they need to be complete. Some of them also have dependencies on other activities (for example, clothes drying can only take place some time after clothes washing), and some of them have several activity profile options that could be chosen to satisfy a requirement (an electric vehicle can be charged fast or slow). Wattage constraints per household might also limit which loads can be activated concurrently without damaging the circuit. Finally, underlying all this there are some exogenous elements:

- Inflexible load, which cannot be controlled by the aggregator, but can be predicted to a certain degree of accuracy, depending on the time of day, day of the week, weather and any special events taking place on that day (such as sports games or festivities.)
- Renewable generation, which depends on the weather conditions, but can also be predicted to a certain degree of accuracy depending on the weather forecast, such as wind direction and speed, cloud cover and solar intensity.
- Wholesale energy costs, which typically change every hour and are determined by the various auction mechanisms of the energy market (Yixing Xu et al., 2010).

These three components have to be modelled as continuous or step functions, which are then overlaid with flexible loads to satisfy the user requirements together with any network stability constraints. The selected plan of activities has an impact on the cost or utility objectives of the aggregator. By using the smart grid infrastructure, it can automate some demand dispatch mechanisms to achieve load-shifting (Pina et al., 2012) and make use of electricity storage facilities. A more detailed mathematical model of this problem is included in the case study presented in Chapter 7.

1.1.1 Approaches for Automated Demand Dispatch

Automated demand dispatch approaches can be classified as *reactive* and *proactive*. *Reactive* approaches involve event-based solutions that queue flexible loads so that they can be actuated in the future when capacity is available. These include algorithms such as Event Driven Scheduling (Busquet et al., 2011) and Reversible Fair Scheduling (Iversen, 2007), which are simple techniques to schedule appliance loads such that the pre-defined power threshold is not exceeded. More sophisticated techniques such as the Earliest Deadline First (Moore, 1968; Kim, 1994; Lehoczky, 1996) and Least Slack First (Barker et al., 2012) algorithms prioritise activities with respect to their deadlines. These techniques react to the current state of the system without looking ahead into the future predicted state.

On the other hand, *proactive* techniques take into account forecasts. Various approaches of this kind have been proposed, such as using stochastic optimisation (Hentenryck and Bent, 2006) and mixed integer linear programming (Scott et al., 2013; Yixing Xu et al., 2010). These approaches can make use of much more information about the power network and the environment it is operating in. Typically, the time-line is split into discrete time-steps, which can introduce scalability issues if the planning horizon is too long or the time-steps are too granular. Also, most of these approaches treat flexible loads as activities that need to be scheduled, without taking into consideration choices about how the activities can be performed, or the optional use of electricity storage facilities.

Automated planning is a good candidate technique for this problem because it also looks ahead to analyse the future states of the system, and tries to find a plan which achieves the required goals within some specified constraints. The techniques proposed in this thesis enhance current planning technology so that it can be used with the temporal and numeric characteristics of this domain. These are incorporated in a planner, which can then be integrated within a wider aggregator system architecture. The prediction data, which includes inflexible demand and generation forecasts together with the wholesale energy costs known for the next planning horizon, is combined with the flexible activities that need to be performed, including their possible load profiles and constraints and electricity storage assets. This formulates the model for the planning problem, which is then solved by the planner.

Automated planning is generic and domain independent. This makes it future proof for any new requirements, such as new types of loads, operational constraints or possible actions that the aggregator can perform, without having to modify the planning system.

1.1.2 Temporal and Numeric Characteristics

The scope of this work is temporal planning problems where, due to *required concurrency*, it is possible that the rate of change of a continuous effect of a *durative action* (Fox and Long, 2003) gets updated during its execution. We refer to this phenomenon as *constants in context*, where rates of change are constant between two discrete time-points (a *context*), but can change in subsequent contexts.

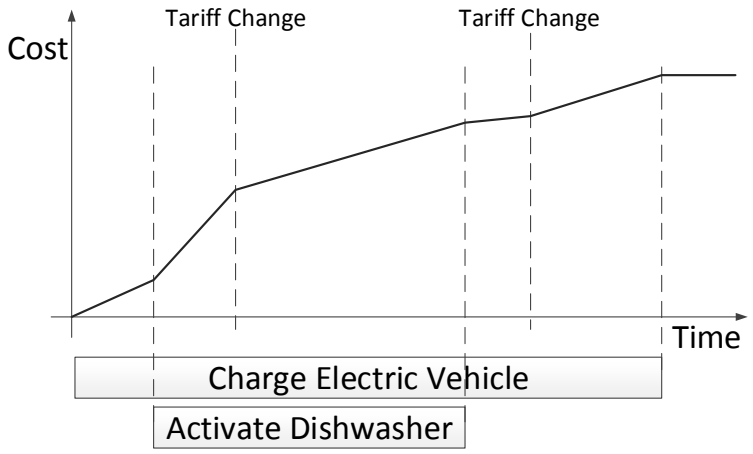


Figure 1.1: Example of a continuous effect with constants in context.

Figure 1.1 illustrates an example of a continuous function, representing cumulative cost of electricity over time, changing its gradient according to various discrete updates, such as an increase or decrease in power consumption (due to an appliance being switched on or off) or a tariff change. The cost of executing an appliance activity is calculated from the wattage rating of the activity, its duration and the electricity tariff active at that point in time. However, the execution of an appliance activity could indeed overlap a tariff change, thus updating the rate of change with which cost increases relative to the time spent executing that activity. While current state-of-the-art temporal numeric planners are capable of handling linear continuous effects, such cases are typically not supported.

The automated demand dispatch problem has a number of characteristics that are not typically found in benchmark planning domains. For this reason, some of these characteristics are not supported by most of the current state-of-the-art temporal and numeric planners, such as POPF (Coles et al., 2010), LPG-td (Gerevini et al., 2004) and TFD (Eyerich et al., 2009). These characteristics are mainly:

Concurrent Actions: The various activities need to be able to run concurrently throughout the plan. This also includes housekeeping activities, such as metering how much energy is being consumed and the cost accumulated so far.

Concurrent Updates: The various activities will have an effect on global numeric state variables, such as the total load in the network.

Piecewise Linear and Non-linear Continuous Functions: Various aspects of the system are intrinsically non-linear. The predicted demand and generation levels together with the step functions of the wholesale energy price are a few examples. Devices such as batteries also exhibit non-linear behaviour.

Flexible Durations: The duration of some activities needs to be decided by the planning system. This is especially the case for activities involving battery charging.

Exogenous Timed Events: Some changes to the system state are not within the control of the planner or plan executive, but are predicted to occur at certain predefined times during the execution of the plan.

Feature	**COLIN/POPF**	**LPG-td**	**TFD**
Concurrent Actions	Yes	Limited	Limited
Concurrent Updates	Yes	No	No
Linear Continuous Functions	Yes	No	No
Piecewise Linear Continuous Functions	Limited	No	No
Non-linear Continuous Functions	No	No	No
Flexible Durations	Yes	No	No
Exogenous Timed Events	Yes	Yes	No

Table 1.1: Temporal and Numeric Feature Support by Current Planners.

Table 1.1 shows the features and limitations of each of the current state-of-the-art temporal and numeric planners. There are other temporal planners, such as SAPA (Do and Kambhampati, 2003a), YAHSP3 (Vidal, 2004) and tBurton (Wang and Williams, 2015), but these focus on other

aspects of temporal planning and are more restricted with regards to the above characteristics. COLIN and POPF do have limited support for piecewise linear continuous effects. They accumulate the rate of change of two overlapping actions that increase or decrease the same numeric variable, but constants in context are not supported. LPG-td and TFD have limited support for concurrent actions. They do not handle all the possible temporal relationships between action endpoints, especially those that interact with the same state variables. The temporal relationships between discrete time points are analysed in further detail in Section 3.2.

The primary focus of this work is to bridge the gap between current planning technology and the requirements for rich temporal and numeric planning domains, such as the problem of planning for automated demand dispatch for an aggregator. This work builds on the algorithm used by COLIN (Coles et al., 2012), which is also used by the planner POPF (Coles et al., 2010). The main difference between these two planners is that POPF identifies independent steps in a plan that do not need to be totally ordered, removes their ordering constraints, and allows them to be scheduled in an arbitrary order, subject to the other constraints. If an action depends on the effect of another action, or effects the same state variable, then this partial ordering does not take place. Since we are in fact focusing on interactions between actions that affect each other's numeric affects, this specific feature is not relevant for this work.

The only other planner known to support most of the required features is the hybrid planner UPMurphi (Della Penna et al., 2009). This planner practically supports the full expressiveness of PDDL 2.1 (Fox and Long, 2002). However, the search mechanism it uses presents a trade-off between scheduling granularity and plan *makespan* (the total time needed to execute a plan). The main reason for this is that it treats time as a sequence of discrete time steps, which imposes practical limits on the makespan of the plan due to the increased branching factor at each step.

The techniques proposed in this thesis treat time as a continuous variable, and only consider discrete time-points where actual state changes take place, improving the scalability dramatically. A comparison between the proposed techniques and UPMurphi (v3.1, the latest version publicly available to date) is included where appropriate throughout this thesis.

1.2 Main Contributions

As shown in Table 1.1, the COLIN algorithm can handle a lot of the requirements, such as concurrency, continuous functions, flexible durations and exogenous timed events. COLIN models the temporal state, obtained from an action of a temporal plan, as a *Simple Temporal Network* (STN) (Dechter et al., 1991), which is then solved in conjunction with the other numeric constraints using a *linear program*. For this reason, the work in this thesis builds on the COLIN algorithm to address the aforementioned limitations described in Table 1.1.

The main contributions of this thesis consist of the following components:

(i) A temporal and numeric planning algorithm to support constants in context.

(ii) A PDDL extension for reusable modules that support non-linear continuous effects.

(iii) An iterative convergence algorithm that enhances the planning system to support non-linear monotonic functions through linear approximations.

A planning system that implements this functionality was developed and the capabilities introduced by these new features are demonstrated with various example domains. This is followed by a case study analysing the planning problem of automated demand dispatch for aggregators, which was the main motivation for this research.

1.3 Thesis Outline

The following is an outline of the subsequent chapters presented in this thesis:

Chapter 2: Background

A detailed description of automated planning concepts is provided in this chapter. The formalism behind classical planning is presented. This is followed by the extensions to support numeric, temporal and hybrid planning together with aspects related to plan quality and metrics. The Planning Domain Definition Language (PDDL) used to model planning problems is also introduced. This chapter concludes with a description of the prevailing data structures and algorithms commonly used to represent planning states, perform reachability analysis, compute heuristics, and model temporal constraints.

Chapter 3: Reasoning with Concurrency

This chapter focuses on the fundamental preliminaries needed to understand the complexity of temporal reasoning when dealing with concurrency. The representation techniques used by most modern temporal planners are described, together with aspects such as continuous change and exogenous timed events. Advanced temporal modelling techniques are also described to demonstrate how temporal relationships between actions can also be modelled in PDDL. This chapter concludes with a detailed description of how scheduling and continuous change are combined together using linear programming to be included as part of the planning process.

Chapter 4: Planning with Constants in Context

This chapter presents an extension to the COLIN algorithm to support linear continuous effects that can be subjected to a modification of their rate of change during their execution. The *temporal relaxed planning graph*, used as a heuristic, is also enhanced to support this new capability. Experiments using some benchmark domains are also provided, to demonstrate the applicability of this technique.

Chapter 5: A PDDL Extension for the Semantic Attachment of External Modules

A PDDL extension is proposed to support the inclusion of external modules. This extension was designed to address various shortcomings in existent semantic attachment or external solver mechanisms. Apart from supporting reusable external modules, this extension also allows for the inclusion of continuous effects that could be non-linear.

Chapter 6: Planning with Non-Linear Continuous Monotonic Functions

This chapter builds on the PDDL extension framework described in Chapter 5, and introduces an algorithm to allow the planner to take into account non-linear continuous functions. These are restricted to continuous functions that are contextually monotonic

between two adjacent discrete states in the plan. This capability opens up various opportunities to model non-linear continuous behaviour more accurately when searching for a plan. It is demonstrated in practice with experiments on some benchmark domains.

Chapter 7: Case Study

The problem of automated demand dispatch is revisited in more detail, bringing in the new modelling and planning techniques proposed in the previous chapters. It is first modelled as a mathematical optimisation problem in terms of inflexible load and generation, flexible load and storage. This representation is subsequently modelled in PDDL, making use of the proposed extensions where necessary. Experiments to evaluate the effectiveness and limitations of the proposed algorithms follow. This chapter concludes with an analysis of the real-world requirements of this domain, presenting ideas about how planning can be applied and used effectively.

Chapter 8: Conclusion

This chapter concludes the thesis with a review of the main findings, together with proposals for future research to follow up from the contributions presented in this work.

2. Background

This chapter introduces the main concepts behind automated planning. The formalism that defines a planning task and a solution plan is described, followed by the extensions necessary to support temporal, numeric, and hybrid planning. A brief description of the modelling language used by most of the current planners, PDDL, is also provided. This chapter concludes with an overview of the prevailing data structures and algorithms used in automated planning, such as the planning graph, heuristic search and local search.

2.1 Introduction to Automated Planning

A *planning task* is a combinatorial problem where a sequence of actions, called a *plan*, is required, such that the *initial state* of a system is transformed into a state that satisfies some *goal condition*. The objective of an automated planning system is to find such a plan.

A planning task is typically defined in the context of a *planning domain* that represents the types of *objects* available within the environment in which the required plan will operate. The domain defines the properties of such objects and the relationships that could exist between them. The domain also defines the *actions* that can be performed to change the state of the system, in terms of these properties and relationships.

A *state*, s, is typically represented as a set of facts. In classical planning these facts are defined using first-order logic (Ghallab et al., 2004). In *numeric planning* this definition is extended with *numeric fluents* that map to a numeric value in that state, while in *temporal planning* this is enhanced even further with information pertaining to timing and scheduling constraints, together with continuous numeric change.

An action is an operator that can be applied to a state, s, to obtain a new state, s'. An action is a triple, $o = \langle name(o), pre(o), \mathit{eff}(o) \rangle$, where:

- $name(o)$ is an expression of the form $a(u_1, ..., u_k)$, where a is a unique *operator symbol*, and $u_1, ..., u_k$ is the parameter list used by the action.
- $pre(o)$ is a conditional expression on the facts of a state, representing the properties of the objects and the relationships between them. For an action, o, to be applicable in a state s, its preconditions must be satisfied, that is $s \models pre(o)$.
- $\mathit{eff}(o)$ is a set of assignment expressions that transform a state, s, into a new state, s', if the action is applicable in s. This state transition is denoted as $s' = \mathit{eff}(o)(s)$.

A *ground* action is an instance of an action where all parameters have been substituted by concrete objects specified by the planning task.

A planning task, P, is defined as a triple, $\langle O, s_0, G \rangle$, where O is the set of ground actions for the task, s_0 is the initial state, and G is the goal condition.

The goal of a planning task, G, is a condition that needs to be satisfied after executing the solution plan. This can be a composite goal made up from a number of sub-conditions. These conditions specify which facts must be true or false, and can include both boolean predicates and also numeric conditions. A goal state, s_G, is a state whose facts satisfy the condition G, written $s_G \models G$, where the $\models$ operator denotes that the state on the left satisfies the condition on the right.

The state transition function of executing an action, o, in a state, s, is thus defined as:

$$\gamma(s, o) = \begin{cases} \mathit{eff}(o)(s) & \text{if } s \models \mathit{pre}(o) \\ \mathit{undefined} & \text{otherwise} \end{cases} \tag{2.1}$$

The *result* of executing a plan $\pi = \langle o_1, ..., o_n \rangle$, starting from a state, s, is defined recursively as follows. The : operator denotes a concatenation of an element to the front of a sequence, with the term on the left representing the *head* (first element) of the sequence, and the term on the right representing the *tail*, that is the rest of the sequence (which could be empty).

$$\mathit{result}(s, \pi) = \begin{cases} s & \text{if } \pi = \langle\rangle \\ \mathit{result}(\gamma(s, x), xs) & \text{if } \pi = x : xs \end{cases} \tag{2.2}$$

Definition 2.1. Given a planning task, $P = \langle O, s_0, G \rangle$, where O is the set of possible ground actions, s_0 is the initial state, and G is the goal condition, a sequence of actions, π, is a plan for P *iff* $\mathit{result}(s_0, \pi) \models G$.

In a classical planning setting the environment is considered to be fully observable, deterministic, static (changes to the system's state only occur through executing an action), and discrete (Russell and Norvig, 2009). Numeric and temporal characteristics do not feature in classical planning tasks.

2.2 The STRIPS Formalism

A classical planning task is commonly expressed using the STRIPS formalism (Fikes and Nilsson, 1971; Lifschitz, 1986). States are represented as a set of logical facts, which are true in that state. Each fact is a first-order literal of the form $p(a_1, ..., a_n)$ where p is a predicate of arity n, with arguments a_1 to a_n. For example, $at(robot1, A)$ denotes the fact that *robot1* is at location A.

A STRIPS state, s, is essentially a conjunction of such facts. Each fact in such a state must be *ground*; that is, each argument must be bound to a concrete object specified in the problem definition. Each first-order literal must also be *function-free*; that is, the arguments of a predicate cannot be derived from the evaluation of a function. STRIPS operates on a *closed-*

world assumption, meaning that any facts that are missing from the state representation can be safely assumed to be false, tying in with the fully observable property of classical planning.

In STRIPS, a condition, C, is a conjunction of a set of function-free positive ground literals. Thus $s \models C$ *iff* $C \subseteq s$. This representation can easily be extended to also include negative literals, with $C = \langle C^+, C^- \rangle$, where C^+ is the set of positive literals and C^- is the set of negative literals of the condition. In this case, a state $s \models C$ *iff* $C^+ \subseteq s \wedge C^- \cap s = \emptyset$. Note that conditions with negative literals are not part of the original set of STRIPS assumptions.

An action's *effect* is a conjunction over a set of function-free literals. These can either add new facts to the state, making them true, or delete existent facts from the state, setting them to false. The set of facts to be added to a state as the result of an action's effect is referred to as the *add list*, also known as the action's *positive effects*, denoted $\mathit{eff}^+(o)$. The set of facts to be removed from a state as the result of an action's effect is referred to as the *delete list*, also known as the action's *negative effects*, denoted $\mathit{eff}^-(o)$. The state transition function for a classical planning problem defined using the STRIPS formalism, $\gamma_c(s, o)$, can be derived from Equation 2.1 as follows:

$$\gamma_c(s, o) = \begin{cases} (s \setminus \mathit{eff}^-(o)) \cup \mathit{eff}^+(o) & \text{if } \mathit{pre}(o) \models s \\ \mathit{undefined} & \text{otherwise} \end{cases} \tag{2.3}$$

Many planners are based on the STRIPS formalism for their internal state representation and action application. Of notable importance are the Graphplan planner (Blum and Furst, 1997) and the FF planning system (Hoffmann and Nebel, 2001), the latter of which is based on the concept of *delete relaxation* (McDermott, 1996; Bonet et al., 1997) to find a *relaxed plan*, which is used as a heuristic to guide the search for a real plan.

2.3 Numeric Planning

While classical planning techniques can be used to solve a broad range of tasks, most realistic problems typically involve accounting for quantifiable resources. For this reason, various initiatives to extend the formalism, languages and planning algorithms were proposed in order to support planning with numeric fluents.

In order to support both propositional and numeric variables, the formalism is extended to carry both types of variables. A numeric planning task, P_n, is defined as $\langle \rho, \vartheta, O, s_0, G \rangle$, where ρ corresponds to the propositional state variables and $\vartheta = \{v_1, ..., v_n\}$ corresponds to the set of numeric variables, also known as numeric fluents. O, s_0 and G represent the set of possible ground actions, the initial state and the goal condition, respectively.

A state, s, is encoded as a pair $\langle P, V \rangle$, where $P \subseteq \rho$ corresponds to the set of propositional variables that are true in that state, and V maps the numeric variables to their respective values, with $V : \vartheta \rightarrow \mathbb{R} \cup \{\perp\}$. An action's preconditions and effects can also include numeric expressions apart from propositional ones. A *numeric expression* can be any arithmetic expression over ϑ and $\mathbb{R}$, using the operators $+$, $-$, $*$ and $/$. $\mathit{vars}(\mathit{expr}) \subseteq \vartheta$ refers to the set of variables appearing in a numeric expression, *expr*.

A *numeric constraint* is a triple, $\langle \mathit{expr}, \bowtie, \mathit{expr}' \rangle$, where *expr* and expr' are numeric

expressions, and $\bowtie \in \{<, \leq, =, \geq, >\}$ is a comparator of the two expressions. A *numeric effect* is also a triple, $\langle v, \otimes, expr \rangle$, where $v \in \vartheta$ is a variable referred to as the *lvalue*, $\otimes \in \{:=, +=, -=, *=, /=\}$ is an assignment operator, and *expr* is a numeric expression representing the right-hand side of the assignment operator, referred to as the *rvalue*. A numeric effect is an *additive* assignment if $\otimes \in \{+=, -=\}$. A numeric effect is a *scaling* assignment if $\otimes \in \{*=, /=\}$ (Fox and Long, 2003).

The precondition, $pre(o)$, of an action, $o \in O$, is a logical expression combining its propositional conditions and its numeric constraints over ϑ, using negation, conjunction and disjunction. $pre^{\rho}(o)$ is the set of propositional variables that feature in $pre(o)$, while $pre^{\vartheta}(o)$ is the set of numeric variables that feature in $pre(o)$.

The action's effect expression, $eff(o)$, is the triple $\langle eff^{+}(o), eff^{-}(o), eff^{\vartheta}(o) \rangle$, where $eff^{+}(o)$ and $eff^{-}(o)$ correspond to the propositional add and delete lists respectively, and $eff^{\vartheta}(o)$ is the list of numeric effects. Furthermore, we define the following sets of numeric variables for the numeric effects of an action:

(i) $lv(o) = \{v | \langle v, \otimes, expr \rangle \in eff^{\vartheta}(o)\}$ refers to the variables that appear as an lvalue in the numeric effects, $eff^{\vartheta}(o)$, of an action, o.

(ii) $lv^{\pm}(o) = \{v | \langle v, \otimes, expr \rangle \in eff^{\vartheta}(o) \text{ and } \otimes \in \{+=, -=\}\}$ corresponds to the $lvalue$ variables of additive assignments in the numeric effects, $eff^{\vartheta}(o)$, of an action, o.

(iii) $rv(o) = \bigcup \{vars(expr) | \langle v, \otimes, expr \rangle \in eff^{\vartheta}(o)\}$ refers to the variables appearing in an rvalue in the numeric effects, $eff^{\vartheta}(o)$, of an action, o.

Note that an action is considered invalid if an $lvalue$ appears more than once in $eff^{\vartheta}(o)$. This ensures that an action does not attempt inconsistent updates on a numeric value, such as assigning two different values to the same variable (Fox and Long, 2003).

Prior to evaluating a numeric expression, *expr*, in a state, s, denoted $expr(s)$, each numeric variable, $v \in vars(expr)$, is substituted by its corresponding value value, $s.V(v)$. If $s.V(v)$ is undefined, or if a division by zero error occurs, the value of $expr(s)$ is $\perp$ (undefined).

A numeric constraint, $\langle expr, \bowtie, expr' \rangle$, holds in a state s, denoted $s \models \langle expr, \bowtie, expr' \rangle$, if both *expr* and $expr'$ are defined in s, and the relationship defined by $\bowtie$ between *expr* and $expr'$ stands, as follows:

$$s \models \langle expr, \bowtie, expr' \rangle \iff (expr(s) \neq \perp \neq expr'(s)) \text{ and } (expr(s) \bowtie expr'(s)) \tag{2.4}$$

The precondition, $pre(o)$, of an action, $o \in O$, holds in a state, s, denoted $s \models pre(o)$, if the evaluation of the logical expression that combines propositional conditions and numeric constraints, using conjunction, disjunction and negation, holds in the state, s.

A numeric effect, $\langle v, \otimes, expr \rangle$, is applicable to a state, s, if $expr(s) \neq \perp$. Furthermore, for the assignment operators $\{+=, -=, *=, /=\}$, which depend on the value of variable v in s, the variable must also be defined in s, that is $s.V(v) \neq \perp$. The effect, $eff(o) = \langle eff^{+}(o), eff^{-}(o), eff^{\vartheta}(o) \rangle$, of an action, o, is *applicable* in a state, s, denoted $eff(o) \rhd s$, if all the numeric effects in $eff^{\vartheta}(o)$ are applicable in s.

The new state, $s' = eff(o)(s)$, is obtained from applying the propositional and numeric

effects of $\mathit{eff}(o)$. The affected numeric variables, $\vartheta_o = \{v | \langle v, \otimes, \mathit{expr}\rangle \in \mathit{eff}^{\vartheta}(o)\}$, will be assigned a new value in s', while the value of the rest of the unmodified fluents, $u \in (\vartheta \setminus \vartheta_o)$, will be copied across from the old state, s.

$$s'.P = (s.P \setminus \mathit{eff}^{-}(o)) \cup \mathit{eff}^{+}(o) \tag{2.5}$$

$$s'.V(u) = s.V(u) \text{ for all } u \in (\vartheta \setminus \vartheta_o) \tag{2.6}$$

$$s'.V(v) = \begin{cases} \mathit{expr}(s) & \text{if } \otimes = := \\ s.V(v) + \mathit{expr}(s) & \text{if } \otimes = += \\ s.V(v) - \mathit{expr}(s) & \text{if } \otimes = -= \\ s.V(v) * \mathit{expr}(s) & \text{if } \otimes = *= \\ s.V(v) \ / \ \mathit{expr}(s) & \text{if } \otimes = /= \end{cases} \quad \text{for all } \langle v, \otimes, \mathit{expr}\rangle \in \mathit{eff}^{\vartheta}(o) \tag{2.7}$$

An action, o, is thus applicable in a state, s, if the state *satisfies* the action's preconditions, $s \models \mathit{pre}(o)$, and its effects are applicable, $\mathit{eff}(o) \triangleright s$. The state transition function for a numeric planning task, $\gamma_n(s, o)$, can be defined as follows:

$$\gamma_n(s, o) = \begin{cases} \mathit{eff}(o)(s) & \text{if } s \models \mathit{pre}(o) \text{ and } \mathit{eff}(o) \triangleright s \\ \mathit{undefined} & \text{otherwise} \end{cases} \tag{2.8}$$

By substituting the generic transition function defined in Equation 2.2 with $\gamma_n(s, o)$ we can define the result of executing a sequence of actions, π, in a numeric planning setting, starting from a state, s, as follows:

$$\mathit{result}_n(s, \pi) = \begin{cases} s & \text{if } \pi = \langle\rangle \\ \mathit{result}_n(\gamma_n(s, x), xs) & \text{if } \pi = x : xs \end{cases} \tag{2.9}$$

An action sequence, π, is a *solution* or a plan for a numeric planning task, $P = \langle O, s_0, G\rangle$, if $\mathit{result}_n(s_0, \pi) \models G$, where G can be composed of both propositional conditions and numeric constraints, combined together in a logical expression using negation, conjunction and disjunction. While this formalism allows an action's precondition and the goal condition to be any logical expression, planning systems typically limit their support to conjunctive goals.

2.4 Temporal Planning

So far, actions have only been discussed in the context of classical and numeric planning. No aspects related to the scheduling of such actions on a timeline were considered. Furthermore, actions were assumed to be instantaneous and sequential. Temporal planning enhances this formalism with time-related characteristics, such as action durations, concurrency, scheduling, and constraints that need to hold throughout the execution of an action. The following formalism follows the same semantics defined in PDDL2.1 (Fox and Long, 2003), adapted to follow through with the same notation of the previous sections.

A temporal planning task, P_τ, is defined as a tuple $\langle \rho, \vartheta, O_{inst}, O_{dur}, s_0, G \rangle$, where:

- ρ is the set of atomic propositional facts.
- ϑ is the set of numeric fluents.
- O_{inst} is the set of grounded *instantaneous actions*.
- O_{dur} is the set of grounded durative actions.
- s_0 is the initial state of the planning task, consisting of a subset of ρ and a mapping of ϑ to numeric values.
- G is the goal condition that needs to be satisfied by a plan.

Instantaneous actions follow the same semantics of actions in classical and numeric planning. An instantaneous action, $o_{inst} \in O_{inst}$, has a precondition expression, $pre(o_{inst})$, that needs to hold in a state, s, for the action to be applicable in that state, and an effect, $\mathit{eff}(o_{inst})$, that transforms a state, s, into a new state, s'. Furthermore if the action has numeric effects, they need to be applicable in s, for the action to be applicable, as defined in Equation 2.8.

A durative action (Fox and Long, 2003) is an action that executes over a time interval rather than at a specific instance. A durative action, $o_{dur} \in O_{dur}$, has the following properties:

- $name(o_{dur})$ is an expression of the form $a(u_1, ..., u_k)$, where a is a unique symbol, and $u_1, ..., u_k$ is the parameter list used by the durative action.
- $startCond(o_{dur})$ is the start condition expression, that has to be satisfied by the state preceding the start of the durative action.
- $endCond(o_{dur})$ is the end condition expression, that has to be satisfied by the state preceding the end of the durative action.
- $durCond(o_{dur})$ is the condition expression, that defines the constraints on the duration of the action, $dur(o_{dur})$. This is a conjunction of a set of temporal constraints of the form $lb \leq dur(o_{dur}) \leq ub$, where $0 \leq lb \leq ub \leq \infty$.
- $inv(o_{dur})$ is the *invariant condition* expression, that defines the constraints that must hold throughout the execution of the action, from the state resulting after applying the start effects of the action, till the state preceding the end of the action.
- $\mathit{startEff}(o_{dur})$ corresponds to the effects that are applied when the durative action commences.
- $\mathit{endEff}(o_{dur})$ corresponds to the effects that are applied when the durative action ends.
- $\mathit{contEff}(o_{dur})$ corresponds to a set of *continuous effects*, with each effect representing continuous change on a numeric variable, $v \in \vartheta$, throughout the action's execution.

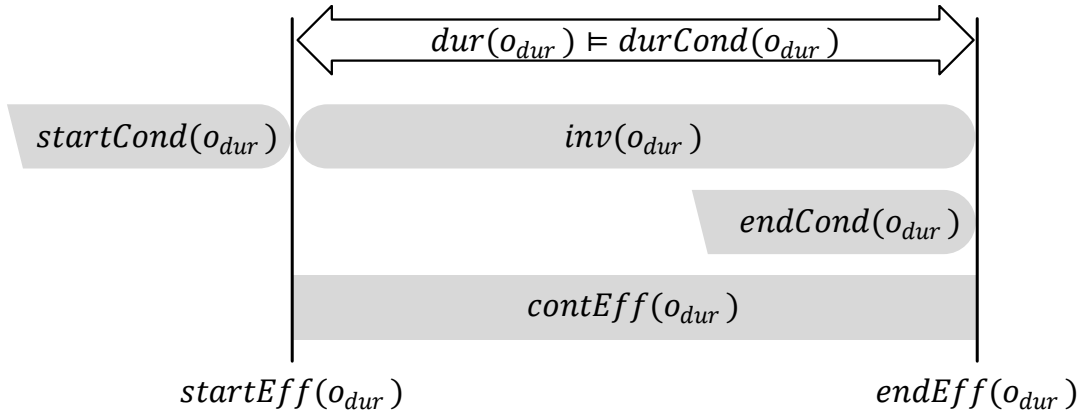


Figure 2.1: Conditions and Effects of a Durative Action.

A *temporal plan*, π_τ, is the tuple $\langle \Theta_{inst}, \Theta_{dur} \rangle$. Θ_{inst} is the list of *timed instantaneous actions*, with each element being a pair, $\langle t, o_{inst} \rangle$, where $t \in \mathbb{R}_{\geq 0}$ corresponds to the time at which the ground instantaneous action, o_{inst}, needs to be executed in the plan π_τ. Θ_{dur} is the list of *timed durative actions*, with each element consisting of a triple, $\langle t, o_{dur}, d \rangle$, where $t \in \mathbb{R}_{\geq 0}$ corresponds to the time at which a ground durative action, o_{dur}, needs to be executed in the plan, π_τ, and d corresponds to the duration chosen for the execution of this instance of o_{dur}. The same ground durative action can potentially be present multiple times in a plan, possibly with different durations.

2.4.1 Continuous Numeric Change

Real-world planning problems often need the capability to reason with continuous change on numeric variables. For example, a planning task could involve vehicles that consume fuel as they travel between two points, and the objective would be to find a plan that is constrained by the amount of fuel available.

Several temporal reasoning frameworks were extended to handle continuous change, such as Event Calculus (Kowalski and Sergot, 1986), which was modified to support continuous change between events (Shanahan, 1990), and ZENO (Penberthy and Weld, 1994), a least-commitment regression planner, whose input language supports continuous change. CSP-based representations were also extended to handle time and continuous numeric change, such as in the case of IxTeT (Trinquart and Ghallab, 2001). Another approach to integrate continuous change with discrete models is hybrid automata (Henzinger, 2000). This formalism was used as the basis for the semantics of PDDL+ (Fox and Long, 2006), an extension to PDDL 2.1 to support processes and predictable events, and also used in planning systems based on model checking (Della Penna et al., 2009, 2012; Bogomolov et al., 2014). More recent techniques involve the extension of SAT-based solvers to support continuous change, such as in the case of TM-LPSAT (Shin and Davis, 2005), and dReal (Bryce et al., 2015). The latter case uses a Satisfiability Modulo Theories (SMT) representation of a temporal numeric planning task with continuous change. It is reduced to a reachability problem of a hybrid system (Henzinger, 2000), with continuous numeric change represented as an ordinary differential equation (ODE). Temporal logics have also been extended to support numeric and continuous change. Mixed

Propositional Metric Temporal Logic (MPMTL) allows formulae to be built over mixed binary and continuous real variables, together with timeline vectors to support continuous change (To et al., 2016).

2.4.2 Exogenous Timed Activity

In the planning framework discussed so far, all the state transitions were controlled by actions selected by the planner. However, real-world problems also involve external state-changing activities that are predicted to happen at certain times during the execution of the plan. Introducing constructs to model such behaviour enriches the planning framework to support concepts such as deadlines and time-windows. There are various real-world examples where such concepts are applicable, such as opening times of an establishment, peak and off-peak hours in a transport network, or daylight time during the day (sunset and sunrise).

Timed Initial Literals (TILs) are a simple way to support the concept of deadlines and time-windows. TILs are essentially facts that become true or false at some predetermined time-points. They are deterministic unconditional exogenous events. Support for TILs was introduced and standardised as part of PDDL 2.2 (Edelkamp and Hoffmann, 2004).

A similar approach can also be used for *Timed Initial Fluents* (TIFs), where instead of modifying a propositional state variable, a numeric fluent is assigned a new value (Piacentini et al., 2015). While this is not officially supported by the standard PDDL, it is implemented by various planners such as COLIN (Coles et al., 2012), POPF (Coles et al., 2010), LPG-td (Gerevini et al., 2004), and UPMurphi (Della Penna et al., 2009). This construct is useful to model changes to numeric values that are predicted to occur during the execution of the plan. Real-world examples include tariff changes of energy, telecommunications or transport services (for instance peak and off-peak rates), predicted UV levels during the day, or tide level forecasts at different times of the day.

2.5 Hybrid Planning

The addition of temporal and numeric constructs to classical planning significantly broadens the applicability of such technology to real world problems. This is especially the case with the introduction of continuous numeric effects, where change is not a discrete stepwise function, but a continuous one evolving in relation to time.

Hybrid planning builds on temporal and numeric planning to model phenomena that occur as a consequence of the state the system is in. The term *hybrid* refers to the fact that a continuous system is controlled by a discrete executive. Hybrid systems can be modelled as Hybrid Automata (Henzinger, 2000), in which discrete-states are also associated with continuous change, captured as *flow conditions*. These correspond to the rate of change of each numeric state variable while in each state. States are also associated with invariant conditions, which cause the system to transition to another state if violated. Discrete state changes are expressed through *jump conditions*. These specify the conditions under which each transition is possible together with its consequences.

Figure 2.2 illustrates a simple example of a hybrid automaton for a thermostat that keeps

the temperature within the bounds $18 \leq x \leq 22$. The system starts in the *Off* state, with a temperature of 20, indicated by the variable x. The flow condition in that state specifies that x decreases with a rate of $-0.1x$ for as long as the system stays in that state. The invariant condition $x \geq 18$ affirms that the state is only valid as long as the temperature is at least 18. The jump condition labelled $x < 19$ indicates that when the temperature goes below 19, a state transition to the *On* state should take place. Similarly, this state defines the rate of change of the temperature, in this case $5 - 0.1x$, and an invariant condition of $x \leq 22$, meaning that the state is only valid for as long as the temperature is at most 22. The jump condition $x > 21$ allows the system to transition back to *Off* when the temperature exceeds 21.

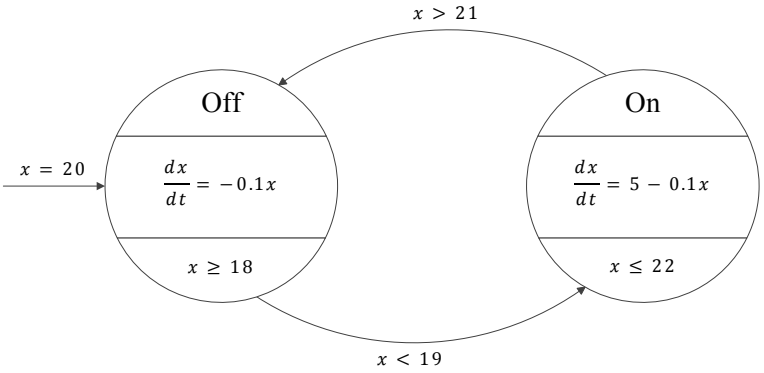


Figure 2.2: Hybrid Automaton for a Thermostat. *Adapted from* Henzinger (2000).

Hybrid planning builds on the semantics of hybrid automata, introducing *processes*, which represent continuous change subject to some invariant condition, and *events*, which trigger a discrete state transition whenever some condition is true (Fox and Long, 2002, 2006). The main difference between these two constructs and conventional planning actions is that processes and events take place as soon as their conditions are satisfied, irrespective of the action choices in the plan. Thus, apart from the explicit action effects, the planner has to account for implicit state changes that will take place as a consequence of these processes and events that could trigger if their conditions are satisfied by the current state.

A *hybrid planning task*, P_H is defined as the tuple $\langle \rho, \vartheta, O_{inst}, O_{dur}, \Phi, \mathcal{E}, s_0, G \rangle$ where:

- ρ is the set of atomic propositional facts.
- ϑ is the set of numeric fluents.
- O_{inst} is the set of grounded instantaneous actions.
- O_{dur} is the set of grounded durative actions.
- Φ is the set of grounded processes.
- $\mathcal{E}$ is the set of grounded events.
- s_0 is the initial state of the planning task, consisting of a subset of ρ and a mapping of ϑ to numeric values.
- G is the goal condition that needs to be satisfied by a plan.

2.5.1 Processes

A process represents a set of continuous effects on one or more numeric variables, subject to some precondition. While in the case of a durative action, the planner can choose the time when the action is applied, or even exclude it from the plan, a process starts immediately when its precondition holds. Unlike durative actions, only one instance of a specific *ground process* can take place at any point in time.

A ground process is a triple $\phi = \langle name(\phi), inv(\phi), contEff(\phi) \rangle \in \Phi$ where:

- $name(\phi)$ is an expression of the form $a(u_1, ..., u_k)$, where a is a unique symbol, and $u_1, ..., u_k$ is the parameter list used by the process.
- $inv(\phi)$ is a conditional expression, that has to be satisfied for the process to be *active* while the system is in a state s, that is $s \models inv(\phi)$.
- $contEff(\phi)$ corresponds to a set of continuous effects of the form $\langle v, expr_\tau \rangle$, with each effect representing a continuous change on a numeric variable, $v \in \vartheta$, throughout the lifetime of the process.

Processes can be seen as an inherent property of the system that manifests itself when it is in a certain state. The executive of the plan's actions has no direct control on a process, apart from performing actions that validate or invalidate its invariant condition. Processes are not reported as part of a plan (Fox and Long, 2006). However, the time points at which a process starts and ends will feature as *happenings* (discrete changes at specific time-points) when the planner is reasoning about the plan. A process, ϕ, can also be active from the beginning of a plan, if its invariant condition is satisfied by the initial state, $s_0 \models inv(\phi)$.

2.5.2 Events

Events represent discrete state changes in the system caused by some precondition becoming true. As with processes, events are not controlled directly by the executive and do not feature in a plan, but control actions can put the system in a state which triggers one or more events. The objective of the planner might be to find a plan that avoids certain events considered bad for the system (Fox and Long, 2002), for example a device overheating, a liquid overspilling its container, or an underwater autonomous vehicle diving beyond its operational depth. On the other hand, the objective of the plan might be to cause a required event, such as reaching the boiling point of a liquid in order to operate some steam powered machinery.

A *ground event* is a triple $e = \langle name(e), pre(e), eff(e) \rangle \in \mathcal{E}$ where:

- $name(e)$ is an expression of the form $a(u_1, ..., u_k)$, where a is a unique symbol, and $u_1, ..., u_k$ is the parameter list used by the event.
- $pre(e)$ is a precondition expression, that will trigger the event if it holds in a state, that is $s \models pre(e)$.
- $eff(e)$ is a set of assignment expressions that transform a state, s, into a new state, s', when the event is triggered. This state transition is denoted as $s' = eff(e)(s)$.

The semantics of events are very similar to those of instantaneous actions, with the fundamental difference being that events trigger as a consequence of their preconditions being satisfied, rather than as a direct consequence of the planner's choices.

Events introduce several challenges. Firstly, an event must be limited from triggering multiple times at the same happening. Otherwise, its precondition would still hold after the effects of the event are applied. Secondly, a state might not just trigger a set of events on their own, but the effects of those events might also trigger others, creating a potentially infinite chain of events if there is a cyclic causal graph between them. Also, applying the sequence of events $\langle e_1, e_2 \rangle$, with both events triggered by a state, s, can yield a different state from the sequence $\langle e_2, e_1 \rangle$. Furthermore, two events triggered by the same state might interfere if they are *mutex*.

In practice, modelling languages for hybrid systems, such as PDDL+ (Fox and Long, 2006), address this by imposing certain rules on the design and semantics of these events. Events are required to invalidate their own preconditions, thus ensuring that they are triggered only once at a happening. Events that trigger concurrently, from the same state, can be restricted to be *commutative*, that is the order in which they are applied will always lead to the same resultant state (Fox et al., 2006), by ensuring they are non-interfering (non-mutex). The conditions for *interference* are defined further on in Definition 3.3. The same restrictions can also be applied to avoid cascading events triggered by the effects of prior ones within the same happening.

2.6 Planning for Dynamic Systems

Apart from the formalisms discussed above, other modelling mechanisms were proposed for reasoning with dynamic systems. Of notable importance is the situation calculus (McCarthy, 1963; McCarthy and Hayes, 1969; McCarthy, 1986; Reiter, 2001), which models the system using actions, situations, and fluents, with respect to objects present in the world being represented. Extensions to it such as the one used in the GOLOG programming language (Levesque et al., 1997) were also proposed.

In the situation calculus, an action, a, has preconditions that determine whether it is *possible* in a situation s, $Poss(a, s)$. An action also has effects, which could be conditional. *Situations* represent the result of applying a finite sequence of actions (Reiter, 1991) in the represented dynamic world, starting from the initial situation, S_0. *Fluents* are statements representing information about a specific situation. These could be boolean predicates, known as *relational fluents*, or *functional fluents* returning a situation-dependent value. Significant research has been done to tackle the *frame* (axioms that are not affected by an action) (McCarthy and Hayes, 1969; Reiter, 1991), *ramification* (indirect effects of actions) (Finger, 1987; McIlraith, 2000) and *qualification* (characterizing the preconditions of an action) (McCarthy, 1977) problems, based on this formalism.

The situation calculus was used to model domains that involve dynamic systems, such as model-based diagnostic problem solving (McIlraith, 1997). It was also used to model *complex actions* (McIlraith and Fadel, 2002), which encapsulate self-contained sub-systems. This representation was successfully translated to PDDL with ADL conditional effects, and used with planning systems such as FF (Hoffmann and Nebel, 2001).

Another commonly used framework is Dynamic Epistemic Logic (DEL) (Gerbrandy and Groeneveld, 1997; Van Ditmarsch et al., 2007). This formalism combines epistemic logic with dynamic semantics, in order to define operations that change the information state of one or more agents.

2.7 Metrics and Plan Quality

In a planning context, the notion of *plan quality* is related to the cost required to achieve the goal from the initial state. A lower cost corresponds to a higher plan quality. The *cost* of a plan, π, in a *classical planning* context (restricted to discrete actions and propositional state variables) is typically the number of actions in the plan, $cost(\pi) = |\pi|$. An *optimal plan* for a classical planning task, is a plan that has the lowest number of actions to achieve the goal from the specified initial state. However, given that STRIPS planning is known to be PSPACE-complete (Bylander, 1994), planners often resort to finding *satisficing* plans rather than optimal plans. This problem becomes even more challenging in temporal planning, where the complexity becomes EXPSPACE-hard (Rintanen, 2007). Furthermore, when time and numbers are introduced, counting the number of actions is not always a good indication of the real cost metric that matters for the planning task.

Definition 2.2. An plan, π^*, is an optimal plan for a planning problem with an initial state, s_0, and a goal condition, G, if $result(s_0, \pi^*) \models G$, and for all other solution plans, $\pi' \in \Pi \setminus \{\pi^*\}$, where $\Pi = \{\pi | result(s_0, \pi) \models G\}$, $cost(\pi^*) \leq cost(\pi')$.

Temporal and numeric planning tasks, together with hybrid planning tasks, typically have an objective function composed of one of the following:

(i) The duration of the plan, also referred to as the makespan of the plan, which corresponds to the time of the last happening.

(ii) An expression that involves the value of one or more numeric variables from ϑ.

The parameters with which plan quality is measured in the context of a planning task are also referred to as *plan metrics* (Fox and Long, 2003). The optimal plan in terms of an objective function which needs to be minimised is $\pi^* = \arg\min_{\pi \in \Pi} cost(\pi)$. If on the other hand the cost needs to be maximised (hence the cost is really a reward function), the objective function becomes $\pi^* = \arg\max_{\pi \in \Pi} cost(\pi) = \arg\min_{\pi \in \Pi} -cost(\pi)$. A weighted multi-objective function that combines makespan with other numeric variables is also possible and supported by some planners, such as Sapa (Do and Kambhampati, 2003b), TPSYS (Garrido and Long, 2004) and LPG (Gerevini et al., 2008). Some systems also apply a technique of building a pareto frontier to try to find the optimal weighting (Sroka and Long, 2012; Khouadjia et al., 2013; Quemy and Schoenauer, 2015).

2.8 Modelling Planning Tasks

In order to model planning tasks in practice, and also to enable interoperability between planning systems (especially for benchmarking and comparison purposes), a Planning Domain Definition

Language (PDDL) was developed. PDDL uses a LISP-style syntax to define facts, variables and action structures, and also Polish prefix notation for expressions.

The initial version (McDermott et al., 1998) supported classical STRIPS and ADL planning (Pednault, 1989, 1994), but was later extended to support numeric and temporal constructs with metrics (Fox and Long, 2003) and timed initial literals (Edelkamp and Hoffmann, 2004), together with a variant to model hybrid systems (Fox and Long, 2006) called PDDL+. Recently PDDL was extended further to support richer constructs such as *plan trajectory constraints* and *preferences* (also known as *soft goals*) (Gerevini and Long, 2005, 2006; Gerevini et al., 2009), and also *object fluents* (finite domain state variables) (Kovacs, 2011). These latter extensions are out of scope for this work. In this section the PDDL syntax will be described briefly, since examples in this thesis also use PDDL.

2.8.1 A Planning Domain Definition Language (PDDL)

In PDDL 1.2 (McDermott et al., 1998), a planning task is typically described using two input files. The first file defines the domain, which includes the *domain-name*, the object *types* (representing types of entities in the real world), *predicates* used to represent facts (possibly related to objects) and actions. It also includes *requirements* to indicate which PDDL features are needed to be supported by the planning system. The domain can define *constants*, objects that are known to exist in each and every possible problem instance. Listing 2.1 shows an example of a domain called `briefcase-world`, defining two types `location` and `physob`, with one constant object `B`, two predicates `atloc` and `in`, and one action `mov-b`. The parameters of predicates and actions are defined using typed variables.

```
(define (domain briefcase-world)
  (:requirements :strips :typing :equality
     :conditional-effects)
  (:types location physob)
  (:constants (B - physob))
  (:predicates (atloc ?x - physob ?l - location)
               (in ?x ?y - physob))

  (:action mov-b
     :parameters (?from ?to - location)
     :precondition (and (atloc B ?from) (not (= ?from ?to)))
     :effect (and (atloc B ?to)
                  (not (atloc B ?from))
                  (forall (?z - physob)
                      (when (and (in ?z B))
                            (and (atloc ?z ?to)
                                 (not (atloc ?z ?from))))))))
     ...
```

Listing 2.1: Example of a PDDL Domain Definition. *Adapted from McDermott et al. (1998).*

The predicate `atloc`, which takes one argument of type `physob` and one of type `location`, represents the fact that a physical object is at a specific location. The predicate `in`, which takes two arguments of type `physob`, represents the fact that one physical object is inside another one.

The action `mov-b`, which represents the operation of moving the constant object `B` from one location to another, takes two arguments of type `location`. The precondition for the action to be applicable is that the predicate `(atloc B ?from)` must be true, representing the fact that `B` is at the first location, and the two locations must not be the same. (The `=` operator is a special construct that checks whether two variables are bound to the same object. This is enabled through the `:equality` requirement.) The effect of the action involves making the fact that the constant object `B` is at the second location and no longer at the first location. It also includes an ADL *conditional effect* (Pednault, 1989, 1994) on all objects `?z` of type `physob`, for which the predicate `(in ?z B)` is true. The location for these objects is also changed accordingly.

The second PDDL input file specifies a problem instance of the domain. This includes the *problem-name*, the domain-name to which the problem instance belongs, the objects (with their respective types from the domain), the initial state (consisting of a set of ground predicates), and the goal state, a logical condition composed of ground predicates combined together using conjunction, disjunction and negation. Listing 2.2 shows an example PDDL problem, named `briefcase-prob1` for the `briefcase-world` domain, with two objects of type `physob` and two objects of type `location`. The initial state, defined by the **`:init`** structure, defines five facts using the two predicates of the domain, `in` and `atloc`. The goal condition, defined by **`:goal`**, is a conjunction of three facts using the `atloc` predicate.

```
(define (problem briefcase-prob1)
  (:domain briefcase-world)
  (:objects pen cellphone - physob
            office home - location)
  (:init (in pen B)
         (in cellphone B)
         (atloc B office)
         (atloc pen office)
         (atloc cellphone office))

  (:goal (and (atloc B home)
              (atloc pen home)
              (atloc cellphone home))))
```

Listing 2.2: Example of a PDDL Problem Definition.

The original PDDL 1.2 specification (McDermott et al., 1998) defined some other rarely used constructs, such as *initial situations*, *expansion* information, and *length*, which were removed in subsequent PDDL versions, namely PDDL 2.1 (Fox and Long, 2003), in order to simplify the language and remove constructs that were generally unused.

A planning system, also referred to as a *planner*, parses these two inputs, validates them, and attempts to find a plan; a sequence of ground actions, that transforms the initial state into a state which satisfies the goal condition.

2.8.2 PDDL 2.1 and 2.2 - Support for Temporal and Numeric Domains

PDDL 2.1 (Fox and Long, 2003) introduced support for temporal and numeric constructs, together with simplifying the language and making it more consistent. Support for numeric fluents was introduced through a separate **`:functions`** block in the domain. These are associated

with a real numeric value rather than a boolean true or false. Listing 2.3 shows an example of some numeric fluents. Some of these, such as the `fuel-level` and `distance` are associated with object types, while others like `total-distance-traveled` and `total-time-traveled` are singleton numeric fluents measuring the total distance traveled and the total time spent travelling by all vehicles, respectively.

```
(:functions
    (fuel-level ?c - car)
    (fuel-capacity ?c - car)
    (velocity ?c - car)
    (distance ?from ?to - location)
    (total-distance-traveled)
    (total-time-traveled))
```

Listing 2.3: Example of Numeric Fluents Defined in PDDL 2.1.

These numeric fluents can be initialised in the problem file as part of the **`:init`** block, as shown in Listing 2.4, corresponding to the values of the numeric fluents in the initial state.

```
(:init
    (= (fuel-capacity car1) 200)
    (= (fuel-level car1) 150)
    (= (velocity car1) 50)
    (= (distance city1 city2) 30)
    (= (distance city2 city3) 40)
    (= (total-distance-traveled) 0)
    (= (total-time-traveled) 0))
```

Listing 2.4: Example of Numeric Fluent Initialisation in PDDL 2.1.

The effect expression of an action was consequently enhanced to support the numeric comparison operations `<=`, `<`, `=`, `>`, and `>=` in preconditions, and updates to numeric fluents in effects, namely **`assign`**, **`increase`**, **`decrease`**, **`scale-up`**, and **`scale-down`**, corresponding to the operators :=, +=, −=, ∗= and /=, respectively. Numeric expressions can include arithmetic operators `+`, `-`, `/` and `*`. The action `refuel`, shown in Listing 2.5, includes a numeric precondition, which checks that the fuel tank is not full, and a numeric effect, which increases the car's fuel-level by the difference between its fuel capacity and its current fuel level.

```
(:action refuel
  :parameters (?c - car)
  :precondition (and (< (fuel-level ?c) (fuel-capacity ?c))
                     (not (driving ?c)))
  :effect (and (increase (fuel-level ?c)
                         (- (fuel-capacity ?c)
                            (fuel-level ?c)))))
```

Listing 2.5: An Action with Effects on Numeric Fluents Defined in PDDL 2.1.

PDDL 2.1 also introduced support for durative actions. Listing 2.6 shows an example of a durative action representing the operation of driving from one location to another. The duration condition states that the duration should be equal to the distance between the two locations divided by the velocity of the car (which is assumed to be constant). The start condition of the durative action states that the car must not already be driving and the car must be at the first

location. The **over all** condition, representing the action's invariants, states that the fuel level must always be above 0 throughout the journey. It also states that at the end, the fuel level must be greater or equal to 0. The durative action also has start effects, which affirm that the car is driving and that it is no longer at the initial location. At the end of the action, the fuel level must be greater or equal to 0. The durative action also has start effects, which affirm that the car is driving and that it is no longer at the initial location. At the end of the action, the fact that the car is at the new location is affirmed, together with the fact that it is no longer driving. Finally, the durative action has three continuous effects, updating the total-distance-traveled, total-time-traveled, and the car's fuel-level in relation to the time spent executing the action. The special #t keyword indicates that the numeric fluent has a continuous rate of change in relation to time, as described in Section 3.3.

```
(:durative-action drive
  :parameters (?c - car ?from ?to - location)
  :duration (= ?duration (/ (distance ?from ?to)
                            (velocity ?c)))
  :condition (and (at start (not (driving ?c)))
             (at start (atloc (?c ?from)))
             (over all (> (fuel-level ?c) 0)
             (at end (>= (fuel-level ?c) 0)))
  :effect (and (at start (driving ?c))
       (at start (not (atloc ?c ?from)))
       (at end   (atloc ?c ?to))
       (at end   (not (driving ?c)))
       (increase (total-distance-traveled) (* #t (velocity ?c)))
       (increase (total-time-traveled) (* #t 1))
       (decrease (fuel-level ?c) (* #t (velocity ?c) 0.1))))
```

Listing 2.6: Example of a Durative Action Defined in PDDL 2.1.

Support for metrics, as a measure of plan quality, was also added in PDDL 2.1. The problem file can include a **metric** directive, which defines the objective function that the planning system should use to determine the quality of a plan compared to another. The **metric** directive can either minimise or maximise the value of an arithmetic expression composed of numeric fluents. Planners supporting PDDL 2.1 are also expected to support a special implicitly defined numeric fluent, **total-time**, representing the makespan of the plan, which can be used in the **metric** directive. Listing 2.7 shows an example of a **metric** directive indicating that a lower value for total-distance-traveled in the goal state is preferred.

```
(:metric minimize (total-distance-traveled))
```

Listing 2.7: Example of a Metric Directive Defined in PDDL 2.1.

Optimal planners (designed to find the best possible plan) or *Anytime* planners (designed to find a satisficing solution as quickly as possible, and improve on it iteratively) can make use of these **metric** directives to find better quality plans.

PDDL 2.2 (Edelkamp and Hoffmann, 2004) built on the constructs introduced in the prior version, and added support for *derived predicates*, predicates that are formulated using other predicates, and timed initial literals, facts that are scheduled to become true or false at a specific point in time, as described in Section 2.4.2. The latter are specified in the **:init** block of the problem file, preceded by the time point at which they are scheduled to become true or false.

Numeric timed initial fluents (TIFs) (Piacentini et al., 2015), which are not officially part of the PDDL specification yet, can also be specified in a similar way, for planners that support them. Listing 2.8 shows an example of both timed initial literals, representing the time-window during which there is daylight, and numeric timed initial fluents, representing the temperature at different times of the day.

```
(:init (at 7 (daylight))
       (at 20 (not (daylight)))
       (at 8 (= (temperature) 18))
       (at 12 (= (temperature) 20.5)))
```

Listing 2.8: Examples of PDDL 2.2 Timed Initial Literals and Numeric Timed Initial Fluents.

2.8.3 Modelling Hybrid Systems with PDDL+

A variant of the mainstream PDDL, called PDDL+ (Fox and Long, 2002, 2006) was developed to support modelling of hybrid systems. This version introduced constructs to define processes and events as described in Section 2.5. Listing 2.9 emulates the thermostat in Figure 2.2, with two events that turn heating *on* or *off* depending on the `temperature`, and two processes that increase or decrease the `temperature` at the specified rates depending on whether heating is *on* or *off*, respectively.

```
(:process heat-water
    :parameters ()
    :precondition (and (<= (temperature) 22) (heating-on))
    :effect (increase (temperature) (* #t (heating-rate))))

(:process ambient-cool-water
    :parameters ()
    :precondition (and (>= (temperature) 18) (not
       (heating-on)))
    :effect (decrease (temperature) (* #t
       (ambient-cooling-rate))))

(:event water-too-cold
    :parameters ()
    :precondition (and (<= (temperature) 19) (not
       (heating-on)))
    :effect (heating-on))

(:event water-too-hot
    :parameters ()
    :precondition (and (>= (temperature) 21) (heating-on))
    :effect (not (heating-on))
```

Listing 2.9: PDDL+ Model of a Thermostat using Processes and Events.

2.9 Planning Algorithms

A planning task is typically represented as a directed graph, with each node representing a possible state of the system, and each edge representing a possible action that leads to a

different state. The edges could also be weighted, to indicate the cost associated with an action. The initial state and any possible goal states are nodes in such a graph. Conventional graph search techniques, such as Breadth-first Search (BFS) or Dijkstra's Algorithm (Dijkstra, 1959) in case of costed edges, can be used for simple planning tasks. However, in practice, the search space is too large to find a plan in an acceptable amount of time. As discussed earlier in 2.7, planning is very hard from an algorithmic complexity perspective. For this reason several heuristic-search or local-search based approaches have been developed over the past couple of decades to help guide the search. In this section the main data structures and search algorithms used in automated planning will be introduced.

2.9.1 The Planning Graph

One of the most significant advances in classical planning was the introduction of a data structure known as the *planning graph* (Blum and Furst, 1997). This has been adopted by most of the current state-of-the-art classical planners in some form or another, and also extended or modified to support numeric and temporal planning tasks. The planning graph acts mostly as a method to perform *reachability analysis* on STRIPS planning domains (Fikes and Nilsson, 1971), to determine whether the goal is actually *unreachable* from the current state. Such a graph does not represent a valid plan, and is typically used to analyse the structure of the problem at hand from the current state. A planning graph helps to discover the relationships between actions and facts, and also identifies which actions are *mutually exclusive*.

A planning graph is a directed graph whose nodes are partitioned into alternating fact and action layers (Blum and Furst, 1997). The first layer, fact layer 0, represents the propositions that are true in the initial (current) state. Each node in each layer can only have an edge to a node in one of the adjacent layers. That is, an action node in action layer 1 can only have edges from one or more proposition nodes in fact layer 0, and edges to one or more proposition nodes in fact layer 1. Similarly, a proposition node in fact layer 1 can only have edges from one or more action nodes in action layer 1, and edges to one or more action nodes in action layer 2. There are three kinds of edges. *Precondition edges* connect nodes in action layer n to nodes in fact layer $n-1$. *Add edges* and *delete edges*, corresponding to the positive effects and negative effects of an action respectively, connect the nodes in action layer n to nodes in fact layer n. Figure 2.3 illustrates the structure of a planning graph.

The planning graph also has *no-op* actions, which propagate all the proposition nodes that appear in any one layer forward to all subsequent fact layers. This makes the cardinality of each fact layer monotonically increasing, and consequently also that of each action layer. A planning graph for a STRIPS planning task whose operators have a constant number of formal parameters, up to a specific number of layers, t, can be created in polynomial time (Blum and Furst, 1997).

The most important characteristic of the planning graph is its capability to optimistically indicate whether a plan could exist, serving as a mechanism for fast reachability analysis. Given a planning graph with t action layers for a planning task P, if a valid plan exists using $s \leq t$ steps, then it must also exist as a sub-graph of the planning graph (Blum and Furst, 1997). Conversely, this means that if the goal propositions are not in fact layer t, a valid plan of length t

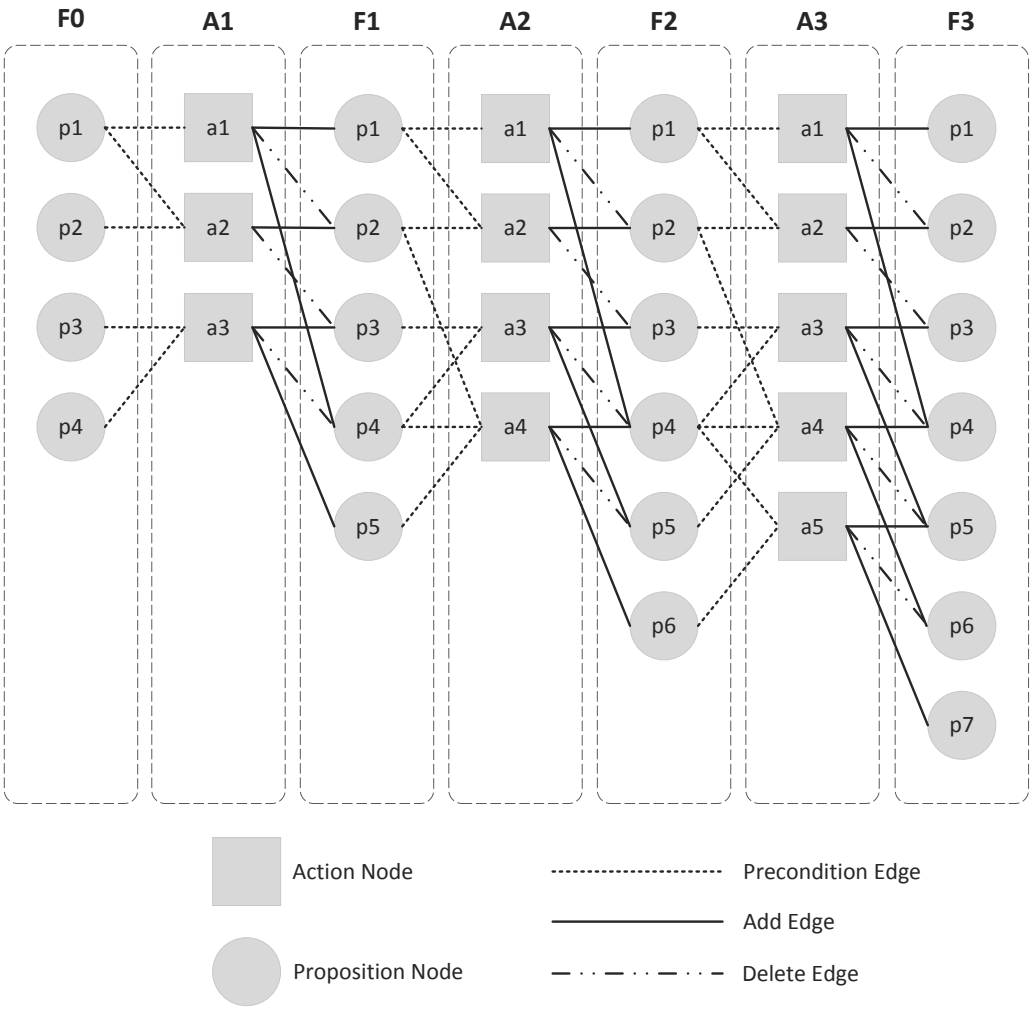


Figure 2.3: A Planning Graph with three action layers. (No-op actions omitted for clarity.)

or less does not exist. The planning graph also indicates actions which interfere with each other (mutex) within the same action layer. It can be used to identify cases where either the effect of one action has an edge to a proposition which is also a precondition edge to the other, or both actions have an effect on the same proposition. Identifying mutual exclusion relationships can help prune out invalid plans. Typically, planning systems propagate and expand the layers of a planning graph until a *fixed point* is reached, that is, no new propositions are added to the subsequent fact layer, and thus any further layer expansions are redundant.

2.9.2 The Relaxed Planning Graph

A simpler variation of the planning graph is the *relaxed planning graph* (Hoffmann and Nebel, 2001). The idea behind this data structure is to generate a planning graph for a *relaxed* version of the planning task at hand where the delete list of each action in the original planning task is ignored, as described in Definition 2.3. This also implies that each edge in the corresponding planning graph of a relaxed planning task can be either a precondition edge from a proposition node to an action node in the next layer, or an add edge from an action node to proposition node in the next layer.

Definition 2.3. For a (classical) planning task $P = \langle O, s_0, G \rangle$, where O is the set of possible actions, s_0 is the initial state, and G is the goal condition, the corresponding relaxed planning task, P', is defined as $P' = \langle O', s_0, G \rangle$, where

$$O' = \{\langle pre(o), eff^+(o), \emptyset \rangle | \langle pre(o), eff^+(o), eff^-(o) \rangle \in O\}$$

Since there are no delete effects, actions happening concurrently will not have any competing needs, and thus no actions will interfere (Hoffmann and Nebel, 2001). Thus, the capability to identify mutual exclusivity is lost in a relaxed planning graph. However, this data structure provides other advantages that can be used during the search process when the planning system explores the state space to find a valid plan. Given that the time to construct a planning graph is polynomial, building it for the relaxed version of a planning task is also polynomial. This makes it very useful to determine which states should be explored first.

The Relaxed Plan

The relaxed planning graph can be used to find a relaxed plan for a relaxed planning task. Since mutually exclusive actions do not feature any more, extracting such a plan is more straight forward. For each goal proposition, g, that is required to be true in the goal condition, G, the earliest layer that makes g true is identified and one of the action nodes in the preceding layer, a_g, that has an add edge to g is selected for that action layer. The preconditions of that chosen action, $pre(a_g)$ are then added to the list of goals to be achieved, such that all goals either have a corresponding achiever action, or are part of the initial state s_0. The relaxed plan is then extracted from the selected actions in each action layer.

Definition 2.4. Given a relaxed planning task $P' = \langle O', s_0, G \rangle$ for a planning task P, the relaxed plan, π', is an action sequence that is a sub-graph of the planning graph of P'.

Of course, given that the negative effects of each action are being ignored, it is unlikely that the relaxed plan is an actual plan for the planning task. However, the length of such a plan, $|\pi'|$, can be used as a *heuristic* to estimate the distance of the current state from the goal.

Helpful Actions

Another characteristic of the relaxed planning graph is its inherent capability to identify which actions will most likely lead to a valid plan from the current state. These *helpful actions* are essentially the nodes that were selected for the relaxed plan from the first action layer of the relaxed planning graph. These actions are considered *helpful* because they achieve one of the goals, or a precondition of another action that is necessary to eventually achieve one of the goals. Helpful actions indicate the promising successors to the current state (Hoffmann and Nebel, 2001), which can help to prune the search space or prioritise which states should be explored next. The set of helpful actions, $H(s)$, for a state, s, and a relaxed planning task with a set of possible actions, O', is defined as follows:

$$H(s) = \{o | s \models pre(o) \text{ and } eff^+(o) \cap G_1(s) \neq \emptyset \text{ and } o \in O'\} \tag{2.10}$$

The term $G_1(s)$ denotes the set of goals that were achieved in the fact layer just after the first action layer (Hoffmann and Nebel, 2001). These goals include preconditions of other actions that feature in the relaxed plan, and thus helpful actions are those considered more useful to get closer to the goal from the current state.

However, in certain cases, using helpful actions to prune out less promising state transitions can also lead to loss of completeness. For this reason, planning systems employing this technique often have a fall-back mechanism to a more complete exhaustive state-space search.

Numeric Variables

The relaxed planning graph has also been extended to support numeric state variables, in addition to propositional ones (Hoffmann, 2003). The concept of delete relaxation is adapted to numeric effects by maintaining a monotonically increasing upper bound and a monotonically decreasing lower bound on each numeric fluent. An action that has a decrease effect will only modify the lower bound, while an increase will only update the upper bound. With each additional fact layer, these bounds drift away further from each other.

In order to determine the point at which the system should stop expanding new layers in the graph, any numeric goal conditions must be evaluated, so that the maximum and / or minimum values needed for a numeric variable are computed. If the upper bound exceeds the maximum needed value, and the lower bound goes below the minimum value for the goal, further expansion is redundant. The relaxed planning graph can then be considered to have reached a fixed point.

However, numeric goal conditions might themselves be defined in terms of more than one numeric state variable. This means that with each new layer, the maximum or minimum goal values would themselves be susceptible to change, heavily impacting the complexity. One approach that is used to circumvent this problem is to increase the upper bound of a numeric variable to ∞ when an action increases its value (Hoffmann, 2003), and similarly decrease its lower bound to $-\infty$ when an action decreases its value. The reasoning behind this approach is that due to delete relaxation, once an action can be applied, it can be executed any number of times, making the numeric fluent's value arbitrarily high or low.

2.9.3 The Temporal Relaxed Planning Graph

The planning graph, and the corresponding relaxed variation described above, do not take into consideration time. There is an implicit ordering relation between action layers, where actions selected from layer n must happen before those chosen from layer $n + 1$. However, no scheduling information is included in this representation. Consequently, variations of the relaxed planning graph were introduced to handle the temporal structure of the planning task. A *temporal planning graph* was introduced by various planning systems. TGP (Smith and Weld, 1999) adopts a simple extension where each action can have a non-negative start time and a positive real-valued duration. Instead of assuming that executing the actions in each layer takes unit time, actions are labelled with the actual start times. Sapa (Do and Kambhampati, 2003a) extends this representation to support richer temporal structures, such as ordering between the start and end *snap actions* (the start and end endpoints of a durative action), and also delete

relaxation. This was refined further in CRIKEY3 (Coles et al., 2008) and COLIN (Coles et al., 2012), fully supporting the temporal semantics of PDDL2.1 (Fox and Long, 2003). This allows different effects and preconditions to be attached to both endpoints of a durative action.

The structure of the Temporal Relaxed Planning Graph (TRPG) is similar to that of the relaxed planning graph, but instead of having each fact layer assigned to an index, it is labelled with a time-stamp that indicates the earliest time when the facts introduced in that layer can become true. For each start snap action introduced to the TRPG, the earliest time that the corresponding end snap action can take place is recorded, computed from the duration constraints of the respective durative action. Snap actions are described in more detail in Section 3.1.

In addition to the propositional and numeric facts holding in a temporal state, COLIN requires two additional pieces of information. The first is a list of durative actions that started prior to the system reaching the current state, but have not finished yet. The second is a set of temporal constraints over the actions in the plan to reach the current state. From this temporal state representation, the TRPG layers are built in a similar way to an RPG, annotating each layer with a time-stamp, and considering both the logical and temporal applicability criteria.

2.9.4 Heuristic Search

Planning is considered to be a very hard problem in terms of complexity. A small planning task with a few entities (objects) and operators (actions) can easily blow up exponentially in the time and space required to perform a complete search to find the best sequence of actions that solve it. For this reason most modern planning systems use some variation of a *heuristic search* (Pearl, 1984) to find a solution in polynomial time. With a good heuristic, *best-first search* algorithms can be very effective in pruning out actions that are less likely to be part of a solution. Best-first search involves computing the *attractiveness* of a node, n, in the search space, through an evaluation function, $f(n)$. This function can use the information associated with the node, the description of the goal, any known information about the domain, the structure of the problem at hand, and potentially information gathered during the search process itself. These algorithms typically maintain a priority queue, with a lower f value indicating a higher priority. Table 2.1 lists some of the most popular heuristic search algorithms.

The effectiveness of these algorithms is of course dependent on the quality of the heuristics used. A heuristic is *admissible* if for every possible node, n, $h(n) \leq c(n)$, where $c(n)$ is the actual distance to the goal. A heuristic is *informative* if for two nodes n and n', $c(n) < c(n') \iff h(n) < h(n')$. While having an admissible heuristic helps to find optimal plans, an informative heuristic is important to provide the appropriate search guidance. One example of an admissible but uninformative heuristic is $h(n) = 0$. Needless to say, finding a generic heuristic that satisfies both criteria and can be computed easily is very hard.

A heuristic is an estimate, or an educated guess, based on the partial information at hand, of the outcome from taking a possible decision. In domain independent planning, heuristics are often functions that extract information from the structure of the problem at hand. This mainly includes the current state, the schema of each action, and the goal definition. The information that can be extracted from just these three data components includes reachability information,

Algorithm	Description	Complete
Greedy Best-First Search	When a node is expanded, each of its successors, n, is evaluated according to an evaluation function $f(n) = h(n)$, where $h(n)$ is a distance estimate of n to the goal. All the successors are put in a priority queue, where the node with lowest estimate is evaluated first.	✓
A* (Hart et al., 1968)	Similar to the greedy best-first algorithm, but also takes into consideration the actual distance of the initial state to the node, n, being evaluated, $g(n)$. The evaluation function is thus $f(n) = g(n) + h(n)$.	✓
Weighted A* (Pohl, 1970)	Uses a weighted evaluation function, $f(n) = g(n) + \epsilon h(n)$, $\epsilon > 1$, relaxing the admissibility criterion to potentially find a sub-optimal solution faster by introducing a bias towards states that are closer to the goal.	✓
IDA* (Korf, 1985)	Iterative Deepening A* is a variant of iterative deepening depth-first search, but using the evaluation function used by A* to determine depth cut-off, increasing it to the f value of the next best excluded node with each iteration. This increases the time required to find a plan (Korf et al., 2001) but decreases the space requirements significantly when compared to A*, since it does not maintain the frontier.	✓
Enforced Hill-Climbing (EHC) (Hoffmann and Nebel, 2001)	Combines the hill-climbing local optimisation technique with breadth-first search. When a node with a strictly better heuristic value than all the other nodes evaluated so far is found, the algorithm immediately expands the better node and ignores the rest of its sibling nodes. While this can be much faster, it is not complete, and planning systems using this approach typically fall back to a variant of A* if no solution is found.	✗

Table 2.1: Best-First Search Algorithms

the causal structure of actions that can lead from the initial state to a goal state, and also the identification of any *landmarks*, that is sub-goals known to be necessary in every plan for the task at hand (Richter and Westphal, 2010). Computing exact information is often NP-hard even for STRIPS planning (Bylander, 1994), so effective methods for computing informative estimates in polynomial time are key. *Delete relaxation* is one of the most popular techniques used to reduce the complexity required to compute an informative heuristic. The optimal cost heuristic, h^+, for the relaxed version of a problem (Hoffmann, 2005) is a lower bound on the optimal cost heuristic, h^*, of the actual problem, making h^+ admissible. However, finding the value of h^+ is also NP-hard (Bonet and Geffner, 2001). Table 2.2 shows some of the most common delete relaxation heuristics used to estimate h^+.

Apart from the heuristics listed in Table 2.2, others variants based on delete relaxation were proposed. These include h^{pmax} (Mirkis and Domshlak, 2007), and h^{sa} (Keyder and Geffner, 2008). However they are mostly special cases or generalisations of the same concepts. h^{CG} (Helmert, 2004), and its successor h^{cea} (Helmert and Geffner, 2008), are also variants of h^{add} adapted for planning with *finite domain* state variables (such as the case of SAS$^+$) instead of just boolean propositions, and take into account causal interactions between such variables. Other heuristics, such as h^{LM} (Richter et al., 2008; Richter and Westphal, 2010), h^L, h^{LA} (Karpas and Domshlak, 2009), and h^{LM-Cut} (Helmert and Domshlak, 2009), also consider landmarks. Finally, abstraction based heuristics such as *pattern databases* (Edelkamp, 2001),

Heuristic	Description	Admissible	Complexity
h^{add} (Bonet and Geffner, 2001)	The *additive* heuristic adds the costs to achieve the individual atoms of the goal together. It assumes that the sub-goals are independent and thus can overestimate the cost when an action achieves more than one goal or sub-goal.	✗	Polynomial
h^{max} (Bonet and Geffner, 2001)	The *max* heuristic computes the cost to achieve each atom of the goal, individually, and takes the maximum value. This heuristic is admissible since the cost of achieving the full set of atoms cannot be lower than the cost of achieving just one of the goal atoms.	✓	Polynomial
h^{FF} (Hoffmann and Nebel, 2001)	Fast Forward (FF) is a delete relaxation heuristic that involves constructing a relaxed planning graph from the current state up to the fact layer which satisfies the goal. The relaxed plan is extracted by extracting the actions that achieve the goal, together with the causal chain of actions that enables them from the initial state. The value of the heuristic function is the length of the relaxed plan. While not admissible, this heuristic has proved to be very informative.	✗	Polynomial

Table 2.2: Delete relaxation heuristics.

merge-and-shrink (Helmert et al., 2007), and *structural patterns* (Katz and Domshlak, 2008), have also shown a degree of success. Since most of these only handle variables that are either propositional or have a discrete finite domain, our focus is on h^{FF}, which has been adapted for both numeric (Hoffmann, 2003) and temporal planning (Coles et al., 2008; Coles et al., 2012).

2.9.5 Local Search

An alternative to heuristic forward search is local search. The idea behind this technique is to explore a *neighbourhood* of possible alterations to the current solution, and choose one of them, performing this repeatedly until one that satisfies the goals and constraints is found. Local search is commonly used in various combinatorial problems, such as discrete optimisation, planning and scheduling. The advantage of this approach is that it intrinsically limits the search space to immediate neighbours. The disadvantage however, is that it can get stuck in local minima. Enforced Hill-Climbing (Hoffmann and Nebel, 2001) is also a local search mechanism.

Local search starts from an initial candidate solution, for which a neighbourhood of alternative solutions could be generated. A *neighbour solution* is another candidate solution obtained by doing a small modification to the current solution. In the context of planning, these could be adding, removing or moving an action in the plan. A neighbour solution is then selected from the neighbourhood according to some criteria and the process is repeated. Local search is an any-time algorithm, which means that if the current solution is already valid one, the process can continue to find a better quality solution, or interrupted to return the current best solution. Local search algorithms often involve a stochastic element to increase the chances of exploring different parts of the search space or improving the chances of escaping local minima. One such technique is Simulated Annealing (Kirkpatrick et al., 1983), which allows the exploration of inferior solutions with a certain probability. Another technique that is also used in classical planning is Monte-Carlo Random Walk (Nakhost and Müller, 2009; Xie et al., 2012; Nakhost

and Müller, 2013), where a set of neighbour states is sampled randomly, before the best one is selected according to some evaluation criteria (typically a heuristic function). Variations of this approach have also been proposed, where once a solution is found, it is optimised using a shortest path or best-first search algorithm, which explores the solution space around the found plan (Nakhost and Müller, 2010).

Local search was also explored in the context of temporal and numeric planning, with the planner LPG (Gerevini et al., 2003c,b,a; Gerevini et al., 2010). This system uses the planning graph to explore possible modifications on the current solution. In this respect, LPG explores the space of possible plans, rather than using forward-chaining to explore the space of possible states reachable from the initial one. Similar to other temporal planners, it splits durative actions into start and end snap actions, and models the constraints as a Simple Temporal Problem (STP) (Dechter et al., 1991), for which a solution must exist. The search prefers plans that solve *flaws*, which could be either unsupported preconditions or temporal flaws involving a negative cycle in the temporal network of the STP. While this planner supports discrete numeric and durative actions, it does not support continuous effects or durative actions with flexible duration conditions. It also offers limited support for concurrency, especially where two concurrent durative actions make use of the same predicate or numeric fluent.

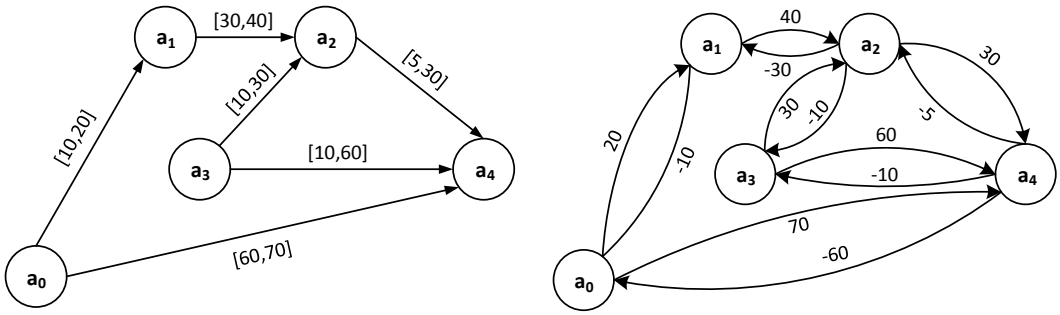
2.9.6 Scheduling and Continuous Change

An important element in temporal planning is scheduling. While a sequence of actions might successfully transform the initial state to one which satisfies the goal conditions, the temporal constraints of the durative actions might make such a plan infeasible. Furthermore, in many domains, there is a preference for plans that are shorter in makespan (the time required to execute the plan) rather than plans with the least number of actions. Current state-of-the-art temporal planning systems, such as COLIN (Coles et al., 2012), POPF (Coles et al., 2010) and LPG (Gerevini et al., 2010) typically use a Simple Temporal Network (STN) to model the constraints of a Simple Temporal Problem (STP) (Dechter et al., 1991).

An STP is restricted to define a *single interval* for the temporal constraints between any two happenings. For two happenings, a_0 and a_1, with the time of each happening represented by $t(a_0)$ and $t(a_1)$ respectively, the temporal constraint between the two happenings is encoded as $l_{0,1} \leq t(a_1) - t(a_0) \leq u_{0,1}$, where $l_{0,1}$ and $u_{0,1}$ represent the lower and upper bounds of the temporal constraint on the time interval between the two happenings.

An STN is a directed acyclic graph, where each node represents a discrete happening in the plan, while each directed edge indicates the temporal ordering between two happenings. Each edge is also labelled with the lower and upper bounds of the temporal constraints on the time that can pass between the two happenings. An STN can also be translated to a directed edge-weighted graph, called a *distance graph* (Dechter et al., 1991). Figure 2.4 illustrates an STP as an STN and its equivalent distance graph.

An STP is *consistent* if and only if it's distance graph has no negative cycles. In order to solve an STP, a shortest path algorithm can be applied to its distance graph to compute the shortest distance between each node and find the minimal network. Finding a solution for this kind of problem is polynomial (Dechter et al., 1991). However, this solution can only be applied



(a) A Simple Temporal Network.

(b) A distance graph for the STN.

Figure 2.4: A Simple Temporal Problem represented as an STN and its distance graph.

when there are no time-dependent numeric variables.

As discussed earlier in Section 2.4.1, durative actions can also feature continuous change; numeric variables that change in relation to the duration of the action. The discrete effects occurring at the start or end endpoints of the action can also have numeric effects expressed in terms of the duration of the action. Furthermore preconditions and invariants can also have constraints on such time-dependent variables. Given that the temporal constraints, numeric preconditions, numeric effects and continuous change can all be modelled as linear inequalities or linear equations, it is possible to combine scheduling with numeric planning and solve a linear program with all the numeric and temporal constraints. Linear programming is also solvable in polynomial time (Khachiyan, 1980; Karmarkar, 1984).

This technique is used by current state-of-the-art planners, such as COLIN (Coles et al., 2012) and POPF (Coles et al., 2010), in order to check the partial plan for consistency in terms of the problem's temporal and numeric constraints, while performing forward-chaining search. This mechanism also computes the value of each numeric variable at each happening and identifies the timestamps of each action in the plan.

2.10 Summary

This chapter provides the formalism of a planning task, which will be used throughout the subsequent chapters. Classical planning is primarily based on the STRIPS planning formalism (Fikes and Nilsson, 1971). This was extended to support numeric planning tasks by introducing rational valued state variables, together with numeric expressions that support arithmetic operations on such variables (Fox and Long, 2003). This framework was augmented further to support temporal planningconstructs, with the inclusion of durative actions; planning operators that take place over a time interval. Durative actions have several additional semantic properties when compared to simple instantaneous actions, such as duration constraints, invariant conditions and continuous effects. An overview of hybrid planning (Fox and Long, 2006) was also provided, analysing how this differs from temporal planning. The planning formalism was subsequently extended further to support processes and events, the main constructs used in hybrid planning.

The concept of plan quality was formalised in terms of the metrics a user of the planning system might be interested in, related to the task at hand. This metric could be some cost or reward objective function associated with executing specific actions, which would be modelled as a numeric expression over one or more of the numeric state variables. It could also be a function on the plan's makespan, the total time required to execute the plan.

A brief overview of the syntax used in the Planning Domain Definition Language (PDDL) was also provided. PDDL is the de-facto standard language used by most of the current planning systems to define planning domains and their corresponding task instances. The various versions of PDDL were described, in terms of the features supported for classical planning (McDermott et al., 1998), temporal and numeric planning (Fox and Long, 2003; Edelkamp and Hoffmann, 2004) and hybrid planning (Fox and Long, 2006).

This chapter concludes with an overview of the data structures and algorithms used in the various flavours of planning. A description of the planning graph is provided, followed by the relaxed planning graph, a simpler representation used to extract heuristic guidance information. The temporal relaxed planning graph is also introduced, which adapts the relaxed planning graph to include temporal information about each layer. A summary of the search mechanisms used in modern state-of-the-art planners is also provided, namely heuristic search and local search, followed by the mechanisms used to model and solve temporal constraints when scheduling the actions in a plan.

3. Reasoning with Concurrency

One of the main challenges in temporal planning is proper support for concurrency. Two or more durative actions can happen concurrently in a plan, and instantaneous actions can also take place during the execution of one or more durative actions, which could potentially also have continuous numeric effects. In this chapter, the prevailing techniques used by current state-of-the-art temporal planners are described in detail. This includes converting each durative action into two instantaneous actions representing its endpoints, an analysis of the temporal relationships between such actions when concurrency is required, support for continuous change and also activities that are prescribed to occur at a specific time. This is followed by a technique that incorporates scheduling within the planning process through linear programming. This chapter also introduces the formalism needed to model temporal planning problems which is used in the rest of the thesis.

3.1 Snap Actions

The structure of a durative action has a lot in common with instantaneous actions. In fact, a lot of temporal planners were built by extending an existent numeric planner, such as Metric-FF (Hoffmann, 2003), Fast Downward (Helmert, 2006), or LPG (Gerevini and Serina, 2002, 2003) and introducing the necessary data structures to support the additional interval constraints.

The most common approach used by several temporal planners, such as LPGP (Long and Fox, 2003b), LPG (Gerevini et al., 2003c), and COLIN / POPF (Coles et al., 2009b, 2010, 2012), is to split a durative action, o_{dur}, into two snap actions, $o_\vdash = start(o_{dur})$ and $o_\dashv = end(o_{dur})$, representing the start and end endpoints of the durative action, respectively. These snap actions have the same structure of instantaneous actions. The precondition of the start snap action is defined as $pre(o_\vdash) = startCond(o_{dur})$, while its effect is defined as $eff(o_\vdash) = \langle startEff^+(o_{dur}) \cup \{p_{o\vdash}\}, startEff^-(o_{dur}), startEff^\vartheta(o_{dur})\rangle$. The precondition of the end snap action is defined as $pre(o_\dashv) = endCond(o_{dur}) \wedge p_{o\vdash}$, while its effect is defined as $eff(o_\dashv) = \langle endEff^+(o_{dur}), endEff^-(o_{dur}) \cup \{p_{o\vdash}\}, endEff^\vartheta(o_{dur})\rangle$. The artificial proposition $p_{o\vdash}$ is a dummy fact that is introduced to ensure that $o_\dashv$ cannot be applied before $o_\vdash$. It is conjuncted with the precondition condition of $o_\dashv$ and added to the add list of $o_\vdash$ and the delete list of $o_\dashv$. The set of operators of a planning task can thus be defined as $O = O_{inst} \cup \{o_\vdash, o_\dashv \mid o_{dur} \in O_{dur}\}$. We will refer to O as the set of *simple actions* (Fox and Long, 2003). This enables us to represent a temporal plan as a sequence of steps, $\langle 0, ..., k\rangle$, at which one or more simple actions take place.

An important note about such snap actions is that $o_\vdash$ and $o_\dashv$ refer to the same *instance*,

$\dot{o}_{dur}$, of a ground durative action o_{dur}. That is $durInst(o_{\vdash}) = durInst(o_{\dashv})$, where $durInst(\dot{o})$ determines the durative action instance corresponding to a snap action, $\dot{o}$. One must keep in mind that the same ground action can reoccur multiple times in a plan, and in temporal planning, where durative actions can run concurrently, two or more instances of the same ground durative action could also execute in parallel. $\vec{o}_{dur}(\pi_\tau)$ refers to the vector of instances of a ground durative action, o_{dur}, where $\vec{o}_{dur}(\pi_\tau) = \langle \dot{o}^i_{dur} \rangle_{i=1...m}$ and m is the number of times o_{dur} appears in a temporal plan, π_τ. For an instance, $\dot{o}$, of a simple ground action o in a plan, $step(\dot{o})$ defines the index of the step at which it is executed in the plan.

The duration constraints, $durCond(o_{dur})$, of a durative action, o_{dur}, can be translated into a set of temporal constraints over its snap actions, of the form $lb \leq ts(j) - ts(i) \leq ub$, where:

- $\{i, j\} \subseteq \mathbb{N}_{>0}$ correspond to the *step indices* in a plan, at which $o_{\vdash}$ and $o_{\dashv}$ take place, respectively. That is $i = step(o_{\vdash})$ and $j = step(o_{\dashv})$.
- lb and ub correspond to the lower-bound and upper-bound of the temporal interval between steps i and j.
- $ts : \mathbb{N} \to \mathbb{R}_{\geq 0}$ determines the time at which a step is to be executed.

The invariant constraints, $inv(o_{dur})$, of a durative action, o_{dur}, need to hold throughout the execution of the action, that is from the time $o_{\vdash}$ is executed, up to the time $o_{\dashv}$ is to be executed. One important subtlety in this model is the fact that $inv(o_{dur})$ needs to hold after $o_{\vdash}$ is executed, but is not a precondition of $o_{\vdash}$. These constraints can be encoded as the tuple $\langle i, j, inv(o_{dur}) \rangle$, which means that the condition $inv(o_{dur})$ must hold for the interval $ts(i) < t_{inv} < ts(j)$, as illustrated in Figure 2.1. The invariant condition can be seen as an additional precondition of the end snap action, $o_{\dashv}$, that holds throughout every state in the plan from when $o_{\vdash}$ is applied. Similarly, the continuous effects of a durative action, $contEff(o_{dur})$, can be encoded as a set of tuples of the form $\langle i, j, ce \rangle$, for each effect, $ce \in contEff(o_{dur})$, that has a continuous effect on a numeric variable over the interval $ts(i) \leq t_{ce} \leq ts(j)$. This follows the PDDL 2.1 semantics that actions have their effects in the interval that is closed on the left, starting at the time at which the action is applied, while preconditions are tested on the interval, that is open on the right, that precedes the action, and invariant conditions are required to hold over an interval that is open at both ends (Fox and Long, 2003).

Another alternative approach, instead of converting durative actions into snap actions, is called *action compression*. This approach is used by TGP (Smith and Weld, 1999), where the Graphplan algorithm (Blum and Furst, 1997) was extended to temporal planning problems, and also MIPS-XXL (Edelkamp and Jabbar, 2006, 2008). The process of action compression involves collapsing a durative action, to a *compressed action* which combines the start, end and invariant conditions as the precondition of the compressed action, and similarly to the effects. However, this approach has two main issues. Firstly, the three sets of conditions can be conflicting in nature. This is also the case for the effects of the durative action. Secondly, this approach is incomplete (Coles et al., 2009c) in cases where concurrency is required and a solution involves the interleaving of the start and end endpoints of two durative actions (Fox et al., 2004). This characteristic is known as required concurrency (Cushing et al., 2007). For

this reason, the snap action approach is considered more complete and superior. On the other hand, action compression might still be useful in some cases where an action is identified to be *compression safe* (Coles et al., 2009a) (that is, action compression does not compromise its soundness and completeness), to minimise unnecessary blow-up in the search space.

In our proposed algorithm we adopt a pure snap action approach, since action compression is not applicable when continuous change is present. It can of course still be used when no continuous change is taking place and an action is compression safe.

3.2 Concurrency

The introduction of durative actions into the planning system presents various challenges. A plan involving durative actions is no longer a totally-ordered sequence of actions, but multiple actions can be required to run concurrently (Cushing et al., 2007).

Definition 3.1. A temporal plan, $\pi_\tau = \langle \Theta_{inst}, \Theta_{dur} \rangle$, is *concurrent* if one of the following conditions hold:

(i) $\exists \{\dot{o}_{dur}, \dot{o}'_{dur}\} \in \Theta_{dur}, \dot{o}_{dur} \neq \dot{o}'_{dur}$, where $ts(step(o_\vdash)) \leq ts(step(o'_\vdash)) \leq ts(step(o_\dashv))$

(ii) $\exists \dot{o}_{dur} \in \Theta_{dur}, \exists \dot{o}_{inst} \in \Theta_{inst}$, where $ts(step(o_\vdash)) \leq ts(step(\dot{o}_{inst})) \leq ts(step(o_\dashv))$

Definition 3.2. A temporal planning problem has *required concurrency* if all possible solutions to the problem are concurrent plans.

The possible temporal relationships of the intervals of two durative actions can be defined using the relations shown in Table 3.1 (Allen, 1983).

Relation	Symbol	Inverse	Description
X before Y	$<$	$>$	X ends before Y starts.
X equals Y	$=$	$=$	X and Y start and end at the same time.
X meets Y	m	mi	X ends at the same time at which Y starts.
X overlaps Y	o	oi	X starts before Y starts, but ends after Y starts and before Y ends.
X during Y	d	di	X starts after Y starts, but ends before Y ends.
X starts Y	s	si	X starts at the same time Y starts, but ends before Y ends.
X finishes Y	f	fi	X starts after Y starts, but ends at the same time that Y ends.

Table 3.1: The possible relations between two intervals X and Y. *Adapted from* Allen (1983).

Some of these relations introduce new challenges to the way durative actions can be ordered such that they guarantee a valid plan for the temporal planning task at hand. The relations of most notable concern are those where one of the interval's endpoints meets an endpoint of another interval; that is the relation is one of $\{=, m, mi, s, si, f, fi\}$. Since the semantics of a durative action allow conditions and effects at both its start and end, two durative actions whose intervals are scheduled in such a way can interfere with each other due to mutex relations between the simultaneous endpoints. The same applies to an instantaneous action scheduled to happen simultaneously with another instantaneous action or an endpoint of a durative action (one of its snap actions). Definitions 3.3 and 3.4 state the conditions for which two simple actions are considered non-interfering or otherwise mutex (Fox and Long, 2003).

Definition 3.3 essentially states that for an action to be non-interfering with the other, it has to: (i) not affect any of the propositions included in the precondition expression of the other action; (ii) not have any contradicting propositional effects with the other action; (iii) not have any numeric effects on variables used by the other action as a numeric precondition or as part of the rvalue of an assignment expression; and (iv) only affect numeric variables that are also affected by the other action if the numeric effects are additive assignments in both actions.

Definition 3.3. Two ground actions, o_1 and o_2, are *non-interfering* if *all* of the following conditions hold:

(i) $pre^{\rho}(o_1) \cap (eff^{+}(o_2) \cup eff^{-}(o_2)) = \emptyset = pre^{\rho}(o_2) \cap (eff^{+}(o_1) \cup eff^{-}(o_1))$.

(ii) $eff^{+}(o_1) \cap eff^{-}(o_2) = \emptyset = eff^{+}(o_2) \cap eff^{-}(o_1)$.

(iii) $lv(o_1) \cap (rv(o_2) \cup pre^{\vartheta}(o_2)) = \emptyset = (rv(o_1) \cup pre^{\vartheta}(o_1)) \cap lv(o_2)$.

(iv) $lv(o_1) \cap lv(o_2) = lv^{\pm}(o_1) \cap lv^{\pm}(o_2)$.

Definition 3.4. Two ground actions, o_1 and o_2 are *mutex* if they are not non-interfering.

For example an action, o_1, with $pre^{\rho}(o_1) = p$ is mutex with an action o_2 whose effect consists of $eff^{+}(o_2) = p$, and also mutex with an action o_3 whose effect consists of $eff^{-}(o_3) = p$, due to point (i) of Definition 3.3. Actions o_2 and o_3 are also mutex, since one action's positive effect is the other's negative effect, violating point (ii). Similarly, an action o_4 with numeric effect $v \mathrel{+}= 10$ is mutex with an action o_5 that has a precondition $v < 10$ and effect $w = v + 3$, since both its precondition and effect make use of v, which is also modified by action o_4, violating point (iii). On the other hand, an action o_6 without preconditions and whose only effect consists of $v \mathrel{+}= 7$ is not mutex with action o_4, even though both modify the same numeric fluent v, since both assignments are additive, complying with point (iv).

When two simple actions are mutex, an explicit ordering can be imposed such that they are not scheduled to execute at the same time. Typically, this is enforced by introducing a small negligible delay, ϵ, between the two mutex actions. This approach is referred to as *non-zero-separation* or ϵ-separation (Fox and Long, 2003; Coles et al., 2012).

A temporal plan, π_τ, with instantaneous and durative actions, consists of a sequence of timestamped steps, known as happenings (Fox and Long, 2003), at which a state transition occurs. The happening sequence $\langle t_i \rangle_{i=0\ldots k}$, for a temporal plan, π_τ, where $t_i \in \mathbb{R}_{\geq 0}$ and $t_0 = 0$, is the ordered sequence of time-points at which discrete state changes take place. $acts(t_i)$ refers to the set of ground simple action instances that are scheduled to take place at t_i, with $acts(0) = \emptyset$. For any time-point $t \geq 0$, $state(t)$ refers to the state resulting from applying the actions at the preceding happening, $\arg\max_i(t_i \leq t)$. Hence, $state(t) = s_0$ (the initial state) for $0 \leq t \leq t_1$ (where t_1 is the happening of the first state transition in the plan).

Another critical aspect to required concurrency is that invariant conditions must hold throughout the execution of a durative action. It is not enough to verify that these conditions hold in the resultant state obtained from applying the start snap action, but they must hold in the state of each subsequent happening until the end snap action is applied. In order to address this requirement, the set of durative actions instances, $exec(t_j)$, that are executing at happening,

t_j, must be identified, such that their invariant conditions, $tinv(t_j)$, which must hold at that happening, can be checked when verifying the applicability of a new action.

Definition 3.5. An instance of a ground durative action, $\dot{o}_{dur}$, is *executing* at a time, t, denoted $\dot{o}_{dur} \in exec(t)$, if there exists a happening, $t_i \leq t$, where $o_\vdash \in acts(t_i)$, and for all subsequent happenings up to t, $o_\dashv \notin \bigcup_{t_i < t_j \leq t} acts(t_j)$.

$$tinv(t) = \bigcup_{\dot{o}_{dur} \in exec(t)} inv(\dot{o}_{dur}) \tag{3.1}$$

For example, the ground durative actions executing at time 10, are those which start at some time before 10, and end at some time after 10. The set conditions that must hold at time 10 corresponds to the union of the invariant conditions of all the actions executing at time 10, as defined in Equation 3.1.

With the above definitions, the applicability of the respective types of simple actions within a temporal plan can be defined as follows:

Definition 3.6. A start snap action, $o_\vdash$, is applicable in a state, s, at a time point, t, denoted $o_\vdash \rhd s$, if $s \models pre(o_\vdash)$, $eff(o_\vdash) \rhd s$, and $\forall cond \in tinv(t) \cup \{inv(\dot{o}_{dur})\}(eff(o_\vdash)(s) \models cond)$.

Definition 3.7. An end snap action, $o_\dashv$, is applicable in a state, s, at time t, denoted $o_\dashv \rhd s$, if $s \models pre(o_\dashv)$ and $eff(o_\dashv) \rhd s$ and $\forall cond \in tinv(t) \setminus \{inv(\dot{o}_{dur})\}(eff(o_\dashv)(s) \models cond)$.

Definition 3.8. An instantaneous action, o_{inst}, is applicable in a state, s, at time t, denoted $o_{inst} \rhd s$ if $s \models pre(o_{inst})$ and $eff(o_{inst}) \rhd s$ and $\forall cond \in tinv(t)(eff(o_{inst})(s) \models cond)$.

Definitions 3.6 to 3.8 state that for an action to be inserted into a plan, the applicability criteria include not only the state prior to the execution of the action, but also whether any invariants that must hold throughout the action's execution time interval were broken. These invariants take into consideration both the invariants of the action itself, and also those of other actions running concurrently, overlapping with the action's execution time interval.

The transition function, $\gamma_\tau(s, o)$, for a temporal planning task (restricted to discrete change), from a state, s, using a simple action, o, can be defined as follows:

$$\gamma_\tau(s, o) = \begin{cases} eff(o)(s) & \text{if } o \rhd s \\ undefined & \text{otherwise} \end{cases} \tag{3.2}$$

3.3 Continuous Change

In our model, the continuous effects, $contEff(o_{dur})$, of a durative action, o_{dur}, are a set of continuous update functions on numeric state variables. A durative action that includes at least one such effect is called a *continuous durative action*. A continuous effect can be represented as $\langle v, expr_\tau \rangle$ where $v \in \vartheta$ is the numeric variable being updated, while $expr_\tau$ is a numeric expression, $expr_\tau : S \to (\mathbb{R}_{\geq 0} \to \mathbb{R})$, corresponding to the effective rate of change on the variable during the durative action's execution.

The rate of change from a continuous effect, $\langle v, expr_\tau \rangle$, on a variable, v, at a time t, can thus be expressed as $expr_\tau(state(t))(t)$. If this is constant throughout the duration spent in a state, the continuous effect is linear.

Definition 3.9. A continuous numeric effect, $\langle v, expr_\tau \rangle$, is *linear* in a state, s, if for all possible values d and d', where $0 \leq d < d'$, $expr_\tau(s)(d) = expr_\tau(s)(d') \neq \perp$.

Definition 3.10. A continuous numeric effect, $\langle v, expr_\tau \rangle$, is *bounded-linear* in a state, s, up to an upper bound duration, ub, if for all possible values d and d', where $0 \leq d < d' \leq ub$, $expr_\tau(s)(d) = expr_\tau(s)(d') \neq \perp$.

The same applicability criteria of discrete numeric effects apply also for continuous effects, extended to the whole duration of the effect.

Definition 3.11. A continuous numeric effect, $\langle v, expr_\tau \rangle$, is applicable in a state, s, denoted by $\langle v, expr_\tau \rangle \rhd s$, if for all possible values of d, where $d \geq 0$, $expr_\tau(s)(d) \neq \perp$.

Definition 3.12. A continuous effect, $\langle v, expr_\tau \rangle$, is *bounded-applicable* in a state, s, up to an upper bound duration, ub, denoted by $\langle v, expr_\tau \rangle \rhd^{ub} s$, if for all possible values of d, where $0 \leq d \leq ub$, $expr_\tau(s)(d) \neq \perp$.

Consequently, the applicability of a continuous durative action also depends on whether its continuous numeric effects are applicable. The duration for a durative action, $dur(o_{dur})$ must also be taken into account since it must satisfy the duration condition, $durCond(o_{dur})$, written as $dur(o_{dur}) \models durCond(o_{dur})$. Due to the possibility of concurrency, intermediate happenings, and thus state transitions, can also occur within the interval of the durative action. This means that not only the numeric effect must be applicable at each intermediate state, but it must also not violate any invariants, including those of other actions running concurrently.

Furthermore, two or more concurrent durative actions could also each have a continuous numeric effect (possibly with the same rate-of-change expression) on the same variable, v. Let $tce(t)$ correspond to the multiset of continuous numeric effects taking place at a time t. This multiset is represented by a set of mappings, $\langle v, expr_\tau \rangle \rightarrow n$, where n is the number of occurrences of the same continuous numeric effect active at time t, defined as follows:

$$tce(t) = \biguplus_{\dot{o}_{dur} \in exec(t)} \{\langle v, expr_\tau \rangle \rightarrow 1 | \langle v, expr_\tau \rangle \in contEff(\dot{o}_{dur})\} \tag{3.3}$$

For clarity, in our representation of a multiset, $m : U \nrightarrow \mathbb{N}_{>0}$, where U is the set of all possible elements, the commutative multiset sum operator, $\uplus$, over two multisets, m_1 and m_2, is defined below, where $Dom(m) = \{a | a \rightarrow n \in m\}$ for a multiset m. The multiplicity operator, #, gives the number of occurrences of an element, a, in a multiset, m, denoted $m\#a$.

$$m\#a = \begin{cases} m(a) & \text{if } a \in Dom(m) \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

$$m_1 \uplus m_2 = \{a \rightarrow (m_1\#a) + (m_2\#a) | a \in Dom(m_1) \cup Dom(m_2)\} \tag{3.5}$$

The cumulative rate of change of a variable, v, at time t, can thus be expressed as follows:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \sum_{\langle v, expr_\tau \rangle \to n \in tce(t)} n \cdot expr_\tau(state(t)) \tag{3.6}$$

Let $\vartheta_f = \{v | \langle v, expr_\tau \rangle \to n \in ceffs\} \subseteq \vartheta$ represent the set of numeric variables affected by continuous change, where *ceffs* is the multiset of continuous numeric effects taking place while in a discrete state. The *numeric flow function*, $flow(s, d, ceffs)$, defines the state, s', obtained from spending time, $d \in \mathbb{R}_{\geq 0}$, in a discrete state, s, with *ceffs* active throughout the whole duration. The values of unaffected variables, $u \in (\vartheta \setminus \vartheta_f)$, together with all propositional variables, stay unchanged, while the values of the affected variables, $v \in \vartheta_f$, change according to a system of differential equations, as defined in Equation 3.6.

Definition 3.13. A numeric condition, *cond*, holds throughout the interval between two consecutive happenings, $t_i \leq t < t_{i+1}$, if $flow(state(t_i), t - t_i, tce(t_i)) \models cond$.

Equation 3.7 defines the notion of *intermediate happenings* occurring within an interval (excluding its endpoints). An interval starting at a specific time point, t, with duration d, is said to have no intermediate happenings, if for a plan's happening sequence, T, $inter(t, d, T) = \langle\rangle$.

$$inter(t, d, T) = \begin{cases} \langle\rangle & \text{if } T = \langle\rangle \text{ or } (T = x : xs \text{ and } t + d \leq x) \\ inter(t, d, xs) & \text{if } T = x : xs \text{ and } x \leq t \\ x : inter(t, d, xs) & \text{if } T = x : xs \text{ and } t < x < t + d \end{cases} \tag{3.7}$$

Definition 3.14. A *temporal context*, $C = [t_i, t_{i+1}]$, in a plan's happening sequence, T, is an interval enclosed by two adjacent discrete happenings, t_i and t_{i+1}, with no intermediate happenings, that is $inter(t_i, t_{i+1} - t_i, T) = \langle\rangle$.

Definition 3.15. The trajectory of the numeric variables affected by a multiset of continuous effects, *ceffs*, starting from a state, s, is valid in relation to a set of conditions, *invs*, throughout an interval with no intermediate happenings of duration, d, denoted $flow(s, d, ceffs) \models invs$, *iff* for all $cond \in invs$ and for all $0 \leq d_c \leq d$, $flow(s, d_c, ceffs) \models cond$.

Definition 3.16. A multiset of continuous numeric effects, *ceffs*, is applicable at a specific happening, t_i, throughout an interval with no intermediate happenings of duration, d, denoted $ceffs \rhd_\tau^d t_i$, *iff* $flow(state(t_i), d, tce(t_i) \uplus ceffs) \models tinv(t_i)$.

Definition 3.16 essentially states that for a multiset of continuous numeric effects to be applicable, the value of the affected variables must not violate any of the invariant conditions throughout the affected period, taking into account any other continuous numeric effects already active. One important aspect to keep in mind is that for a continuous numeric effect, $\langle v, expr_\tau \rangle$, the expression, $expr_\tau$, representing the rate of change of variable v, can itself include continuously changing variables, making the effect non-linear.

Definition 3.17. A durative action, o_{dur}, is applicable at a time point, t, denoted $o_{dur} \rhd_\tau t$, if there exists $d \models durCond(o_{dur})$ where $o_\vdash \rhd state(t)$ and $o_\dashv \rhd state(t + d)$, and for the sequence of happenings $\langle t_i \rangle_{i=0..k} = t : inter(t, d, T) : t + d$, with T being the sequence of all happenings in the current plan, $contEff(o_{dur}) \rhd_\tau^{t_{j+1} - t_j} t_j$ for all $i \leq j < k$, $j \in \mathbb{N}$.

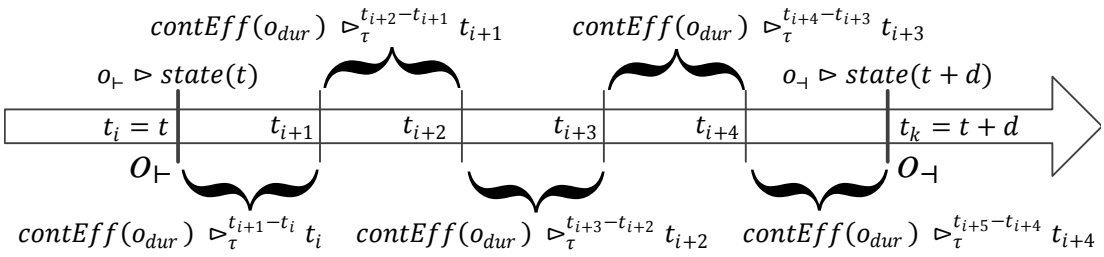


Figure 3.1: Applicability of a durative action with intermediate happenings.

Figure 3.1 illustrates a durative action, o_{dur}, with duration $dur(o_{dur}) = d$, being applied at time t, with intermediate happenings $\langle t_{i+1}, t_{i+2}, t_{i+3}, t_{i+4} \rangle$. Planners that use forward-search, such as COLIN (Coles et al., 2009b, 2012), typically do not need to check any intermediate happenings, since new simple actions are always appended to the end of the plan.

From the above definitions, we can now derive the *state flow function*, $\xi(s, d, ceffs, invs)$, where s is the starting discrete state, $d \in \mathbb{R}_{>0}$ is the duration spent in the state, *ceffs* is the multiset of continuous effects (a set with elements of the form $\langle v, expr_\tau \rangle \rightarrow n$), and $invs$ is the set of invariant conditions that must hold throughout the period spent in state s.

$$\xi(s, d, ceffs, invs) = \begin{cases} flow(s, d, ceffs) & \text{if } flow(s, d, ceffs) \models invs \\ undefined & \text{otherwise} \end{cases} \tag{3.8}$$

The state flow function, ξ, can be combined with the *timed transition function*, γ_τ, defined in Equation 3.2, to reflect the full behaviour of the system executing a plan with happening sequence, T, that includes continuous numeric effects. The $apply(s, \pi_t)$ function, where s is a state and π_t is a sequence of simple actions designated to execute at a happening, t, defines the state transition of applying the discrete effects of such actions.

$$apply(s, \pi_t) = \begin{cases} s & \text{if } \pi_t = \langle\rangle \\ apply(\gamma_\tau(s, x), xs) & \text{if } \pi_t = x : xs \end{cases} \tag{3.9}$$

The result of executing a plan's sequence of happenings, T, at their respective timestamps, starting from a state, s, is defined by $result_\tau(s, T)$. The operator seq[] converts a set into an arbitrary sequence of the same size that includes all the elements of the set.

$$result_\tau(s, T) = \begin{cases} s & \text{if } T = \langle\rangle \\ apply(s, \text{seq}[acts(t)]) & \text{if } T = \langle t \rangle \\ result_\tau(\xi(apply(s, \text{seq}[acts(t_0)]), & \\ \qquad t_1 - t_0, tce(t_0), tinv(t_0), t_1 : ts) & \text{if } T = t_0 : t_1 : ts \end{cases} \tag{3.10}$$

3.4 Timed Happenings

As discussed earlier in Section 2.4.2, temporal planning also involves exogenous timed activity. This is modelled in PDDL2.2 as timed initial literals (TILs) (Edelkamp and Hoffmann, 2004; Hoffmann and Edelkamp, 2005), facts that become true or false at a specific time, or timed initial fluents (TIFs) (Piacentini et al., 2015), representing a discrete update (assignment) to a numeric variable at a specific time. We refer to discrete state changes resulting from TILs and TIFs as *timed happenings*, that is discrete state changes scheduled to happen at a specific time during the execution of the plan. Timed happenings have two intrinsic properties:

(i) They have an explicit temporal constraint specifying exactly the time stamp when they are bound to take place.

(ii) They will always feature in the plan, as long as the plan's makespan is long enough to incorporate the time when they are bound to take place.

These two characteristics bring in new dynamics to the search process. In this work, we compile TILs and TIFs into actions without preconditions, that are bound to happen at some point in time during the plan. The applicability of the other actions needs to consider these extra discrete changes that are to take place, and the search process needs to plan around them.

We model a timed initial literal is a pair $\langle t, l \rangle$, where $t \in \mathbb{R}_{>0}$ corresponds to the time at which l will hold, and l is a literal that represents the affirmation (or negation) of a propositional state variable, p (or $\neg p$). A timed initial literal affirming a proposition, p, at timestamp t, is compiled into a simple action, o_{til}, with $pre(o_{til}) = true$, $\mathit{eff}^-(o_{til}) = \mathit{eff}^{\vartheta}(o_{til}) = \emptyset$, and $\mathit{eff}^+(o_{til}) = \{p\}$, occurring at a happening, t. Conversely, a timed initial literal negating a proposition, $\neg p$, at a time point, t, is compiled into a simple action, o_{til}, with $pre(o_{til}) = true$, $\mathit{eff}^+(o_{til}) = \mathit{eff}^{\vartheta}(o_{til}) = \emptyset$, and $\mathit{eff}^-(o_{til}) = \{p\}$ occurring at a happening, t.

We model a numeric timed initial fluent as a triple $\langle t, v, x \rangle$, where $t \in \mathbb{R}_{>0}$ corresponds to the time at which $v \in \vartheta$ will be set to the value $x \in \mathbb{R}$. We compile a numeric timed initial fluent into a simple action, o_{tif}, with $pre(o_{tif}) = true$, $\mathit{eff}^+(o_{tif}) = \mathit{eff}^-(o_{tif}) = \emptyset$ and $\mathit{eff}^{\vartheta}(o_{tif}) = \{\langle v, :=, x \rangle\}$ occurring at a happening, t.

The same notion of non-interference and mutex actions defined in Definitions 3.3 and 3.4 apply to both TILs and TIFs, when considering the applicability of other actions to be scheduled to execute at or during the time of a timed initial literal or numeric timed initial fluent.

Let $H = \langle t_i \rangle_{i=0..k}$ be the sequence of timed happenings where $\mathit{tils}(t_i) \cup \mathit{tifs}(t_i) \neq \emptyset$, with $\mathit{tils}(t)$ corresponding to the set of TILs scheduled to take place at time t, and $\mathit{tifs}(t)$ corresponding to the set of TIFs scheduled to take place at time t. Assuming that the TILs and TIFs for a specific timed happening, t, are non-interfering with each other they can be compiled into one action, o_t, where $pre(o_t) = \emptyset$, $\mathit{eff}^+(o_t) = \{\mathit{eff}^+(o) | o \in \mathit{tils}(t)\}$, $\mathit{eff}^-(o_t) = \{\mathit{eff}^-(o) | o \in \mathit{tils}(t)\}$, and $\mathit{eff}^{\vartheta}(o_t) = \{\mathit{eff}^{\vartheta}(o) | o \in \mathit{tifs}(t)\}$. Assuming that o_t is non-interfering with any of the other simple actions executing at t, $\mathit{acts}(t)$, the transition function for a happening, t, can be defined as follows, where s is the state prior to time t.

$$s' = \mathit{apply}(s, o_t : \text{seq}[\mathit{acts}(t)]) \tag{3.11}$$

In order to support timed happenings in the search for a valid plan, extra steps for each TIL or TIF are added to the plan, π. The step, i, of a timed happening, t, also needs to have the corresponding temporal constraint added to the set C, taking the form $ts(i) = t$, which can be translated into $t \leq ts(i) \leq t$. The search process then needs to consider the different ordering of actions in conjunction with timed happenings, to find a valid plan.

3.5 Advanced Temporal Modelling

Temporal planning problems often feature advanced timing constraints or resource access dependencies. More specifically, a sophisticated temporal domain often features:

(i) An action needs to start exactly after the end of another action.

(ii) An action needs to happen after a minimum time interval from another action.

(iii) One or more actions need to happen during the execution of another durative action.

(iv) An action needs to refer to the time-point at which it is executed to compute the level of a resource that is being produced or consumed in relation to time.

These requirements can be achieved with higher order constructs built on the propositional and numeric temporal planning framework described so far. The following are a selection of such constructs, namely *clip actions*, which are used to force two actions to execute within a certain time interval, *strut actions*, which force two actions to separate for a minimum time interval, and *envelope actions*, which wrap around one or more actions in a plan. These constructs can be used to achieve the above modelling requirements.

3.5.1 Clip Actions

Clip actions (Fox et al., 2004; Coles et al., 2009a; Coles and Coles, 2014) are durative actions whose purpose is to enforce a maximum separation between two actions. This minimum separation can be as small as required, to force one action to take place after another, taking into consideration ϵ-separation.

The intuition is to introduce an auxiliary proposition, c, which is a precondition for the first action's end endpoint, and also the second action's start endpoint. This precondition would only hold during the execution of the clip action. Let *ClipAB* be a clip action, where a is a start condition of *ClipAB*, $startEff^{+}(ClipAB) = \{c\}$, and b is an end condition of *ClibAB* and $endEff^{-}(ClipAB) = \{c\}$. An action A, which maintains the fact a throughout its execution will force *ClipAB* to start before A ends. $endCond(A)$ also includes c as its condition, which forces *ClipAB* to be included in the plan. Finally, action B, which needs to be clipped to A, will also have c as part of $startCond(B)$, and $b \in startEff^{+}(B)$ to force *ClipAB* to finish after its start endpoint. This is illustrated in Figure 3.2, where conditions are shown on top of an action, and effects are shown at the underneath it, attached to the corresponding endpoint.

The same concept can also work for actions that need to take place at a specific absolute time from the start of the plan. Instead of a clip action, two TILs can be used. The first one affirms the auxiliary proposition, c at a time, $t - \epsilon$, and a subsequent one affirms $\neg c$ at $t + \epsilon$.

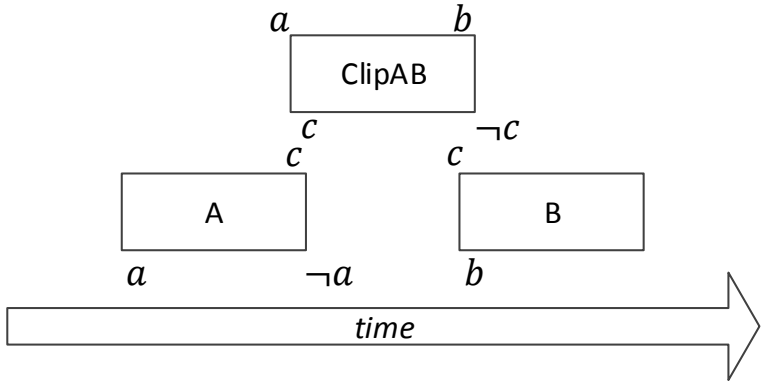


Figure 3.2: A *clip action* forcing two actions to be clipped together.

This way the timed action can only take place in the period determined by the tight clip between $t - \epsilon$ and $t + \epsilon$ (which due to ϵ-separation will be at t).

3.5.2 Strut Actions

A strut action enforces a minimum time interval between two actions (Fox et al., 2004). It is also a durative action, with its duration condition specifying the required minimum interval. A strut action is associated with two auxiliary propositions s and s_2. $s1$ is added by the first action and is an invariant condition of the strut action S. s_2 is added at the end of the strut action and is a precondition to the start of the second action. In order to control the bound on the specific separation interval clip actions can be used with the strut action to tie the three actions together, as illustrated in Figure 3.3.

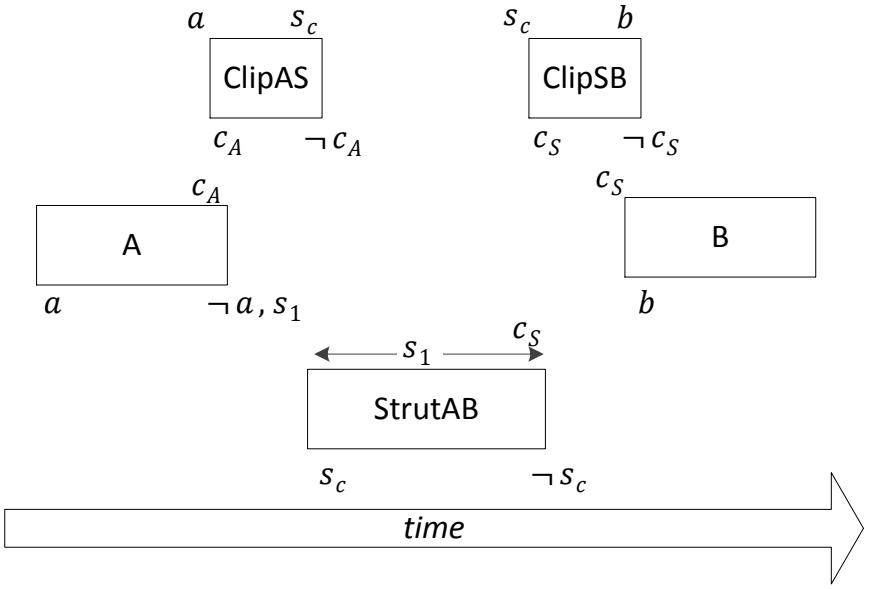


Figure 3.3: A *strut action* forcing two actions to be separated apart.

3.5.3 Envelope Actions

An envelope action (Coles et al., 2009a) can be used to wrap around a number of actions in the plan. It can be used to force some sequence of actions to take place within a certain duration, or also model resource availability and continuous consumption in relation to time. An envelope action makes use of one auxiliary proposition, c, which is needed as an invariant condition by any other actions that need to be enclosed within the time envelope. Figure 3.4 illustrates this with an envelope action *EnvelopeAB*, that affirms c at its start, and negates c at its end, forcing actions A and B to take place during its execution, since both have c as their invariant condition.

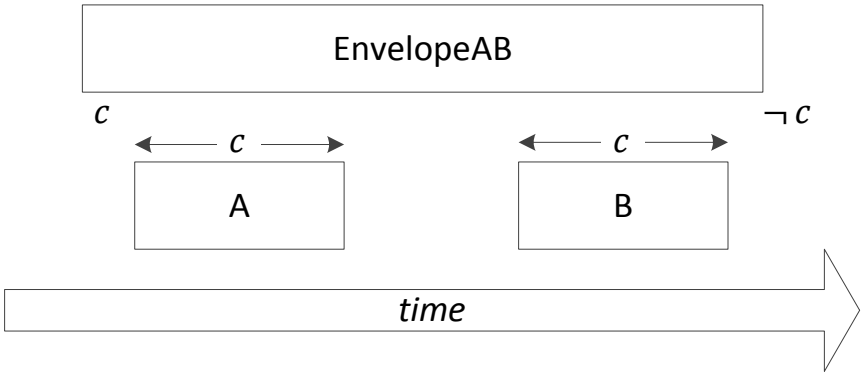


Figure 3.4: An *envelope action* containing two actions.

A special application of this construct is to model time as a continuous effect of the envelope action, which is forced to start at the beginning of the plan, and end after all the actions of the plan terminate. This special envelope action, `timer`, would have a continuous effect on `current-time`, which increases in relation to the elapsed time in the plan. Other actions in the plan can refer to `current-time` if they need to determine the time-stamp at which one of their endpoints takes place and use it for one of their conditions or effects.

3.6 Planning and Scheduling with Linear Programming

In order to find a valid temporal plan for a problem with continuous effects, the assignment of time-dependent numeric fluents must be performed in conjunction with the scheduling process. The approach described below is similar in spirit to the one used by COLIN (Coles et al., 2009b), where a Linear Program (LP) is used to find a feasible assignment to the numeric fluents and the timestamps of each happening (Fox and Long, 2003), such that all the action preconditions and temporal constraints are satisfied. However, there are some key differences in the way numeric fluents are represented and effects are propagated throughout the plan. COLIN stands for *COntinuous LInear Numeric* change, which was the main innovation introduced by the planner carrying the same name. The key intuition behind COLIN is that the numeric and temporal constraints of the planning task can be modelled as a linear program and solved simultaneously. The algorithm performs a best-first forward-chaining search, starting from the initial state. It constructs the plan by appending new simple actions (instantaneous or snap actions), and after each step it builds a linear program to test it for consistency.

A linear program (LP) consists of variables and linear constraints on these variables. These constraints can be inequalities of the form $expr \geq 0$. The expression $expr$ has to be linear, of the form $a_1x_1 + a_2x_2 + ... + a_nx_n$, where x_i is a variable to be solved by the linear program and a_i is a constant coefficient associated with that variable. A linear program typically also has an objective function, which it would try to minimize (or maximize), in terms of these variables. In order to distinguish between the variables in the LP and numeric state variables of the planning task, we will refer to the latter as numeric fluents.

When applying an LP to a temporal numeric planning task, the variables of the LP will correspond to:

- the timestamps of each simple action,
- the values of the numeric fluents before each happening, and
- the values of the numeric fluents after each happening

On the other hand, the constraints of the LP correspond to:

- the temporal constraints of each durative action (including any ordering relationships implicit from the plan)
- the numeric preconditions of each simple action,
- the numeric discrete effects of each simple action (including TIFs),
- the numeric invariant constraints of each durative action (that need to hold at each happening between both start and end snap actions), and
- the numeric continuous effects of each durative action.

The objective is typically set to minimise the variable representing the timestamp of the last happening, thus minimising the makespan of the plan. However, if a plan quality metric is defined for the planning task, as described in Section 2.7, the LP's objective function can be set to correspond to the respective numeric fluents that reflect cost or utility of the plan. Nevertheless, this approach does not guarantee cost optimality, because the LP is still restricted to the action ordering specified by the plan.

Given a temporal numeric planning task $P_\tau = \langle \rho, \vartheta, O, s_0, G \rangle$, as defined in Section 2.4, with $O = O_{inst} \cup \{o_\vdash, o_\dashv \mid o_{dur} \in O_{dur}\}$ as defined in Section 3.1, a plan π_τ for P_τ is found by performing a forward search from the initial state and applying actions in O, obtaining new states from the effects of each of the applied actions. After appending a new action to the plan, the LP is executed to verify temporal and numeric consistency for the actions and *time dependent* numeric fluents.

To illustrate this concept, consider the following example:

- A numeric fluent, v, set to the value of 0 at the initial state.
- A durative action a, with $durCond(a) = \{(5 \leq dur(a) \leq 8)$, and $contEff(a) = \langle v, 1 \rangle$.

- A durative action b, with $durCond(b) = \{(4 \leq dur(b) \leq 10)\}$, $startCond(b) = \{(3 \leq v)\}$, $endCond(b) = \{(v \leq 9)\}$ and $endEff(b) = \{(v \mathrel{+}= 3)\}$.
- A timed initial literal, $\langle 7, l \rangle$, stating that l should take place at time-point 7.

Let us consider the case where the planner is evaluating the totally ordered sequence $\langle a_\vdash, b_\vdash, l, b_\dashv, a_\dashv \rangle$. In this scenario, a is an envelope action with a continuous effect on v throughout the plan. An LP can be formulated for this action sequence with the variables $\langle x_1, ..x_{15} \rangle$, where x_1 to x_5 correspond to the time for each step, s_1 to s_5, in the plan, each x_{2i+4} corresponds to the value of v before each step s_i, referred to as v_i, and each x_{2i+5} corresponds to the value of v after each step s_i, referred to as v'_i.

If the aim is to minimise makespan, the LP would be formulated as follows:

Minimise: x_5	
Subject to:	
$0 \leq x_1$	$0 \leq a_\vdash$
$x_1 + \epsilon \leq x_2$	$a_\vdash \prec b_\vdash$
$x_2 + \epsilon \leq x_3$	$b_\vdash \prec l$
$x_3 + \epsilon \leq x_4$	$l \prec b_\dashv$
$x_4 + \epsilon \leq x_5$	$b_\dashv \prec a_\dashv$
$5 \leq x_5 - x_1 \leq 8$	$durCond(a)$
$7 = x_3$	$\langle 7, l \rangle$
$4 \leq x_4 - x_2 \leq 10$	$durCond(b)$
$0 = x_6$	v_1
$x_6 = x_7$	v'_1
$x_7 + x_2 - x_1 = x_8$	v_2
$x_8 = x_9$	v'_2
$x_9 + x_3 - x_2 = x_{10}$	v_3
$x_{10} = x_{11}$	v'_3
$x_{11} + x_4 - x_3 = x_{12}$	v_4
$x_{12} + 3 = x_{13}$	$v'_4 : endEff(b)$
$x_{13} + x_5 - x_4 = x_{14}$	v_5
$x_{14} = x_{15}$	v'_5
$3 \leq x_8$	$startCond(b)$
$x_{12} \leq 9$	$endCond(b)$

The first group of constraints enforce the total ordering between each step using ϵ-separation (the precedence operator $\prec$ represents a temporal ordering constraint)[1]. The second group of constraints model the temporal conditions of the actions and timed initial literal. The third group of constraints represents the trajectory of numeric fluent v throughout the steps in the plan. It starts from its value in the initial state, and is carried forward throughout the plan, applying the respective numeric effects to it at each step. x_6 represents the value of v at the initial state, before the first action is applied. x_8, x_{10}, x_{12} and x_{14} apply the continuous effect of a on v, which increments it with rate of change 1, depending on the interval between each intermediate adjacent step. x_{13} applies the discrete numeric effect $endEff(b)$, which increases v by 3. The other equalities are responsible for carrying forward the value of v. The fourth group of constraints represents the two start and end preconditions of durative action b.

With $\epsilon = 0.001$, the result of this LP would be plan schedule $\langle 0, 3, 7, 7.001, 7.002\rangle$, with the plan ending 2ϵ after the timed initial literal l, and the final value of $v = 10.002$.

3.7 Time Dependence

Identifying the fluents that actually need to be present in the LP can be critical to achieve a scalable performance. In fact, the authors of the COLIN planner introduce the notion of a *stability*. A numeric fluent, v, only needs to be included in the LP for steps after which it has become *unstable*, that is there was a direct continuous numeric change on v, there was a direct duration-dependent change on v, or there was a discrete change where the magnitude of the change was calculated from an expression that involved an unstable variable (Coles et al., 2012). We redefine this notion of stability to that of time-dependence at a specific step k. Once a numeric fluent is assigned to a non-time-dependent value, it becomes non-time-dependent again for the subsequent steps. We also allow discrete numeric effects that are not supported by the LP (they are non-linear) on variables at the steps where they are non-time-dependent.

Definition 3.18. A numeric fluent evaluated at step k of a plan is *time-dependent* at k if either:

(i) it is modified by a continuous effect from a durative action, a_{dur}, where $a_\vdash$ takes place at step $i < k$, or

(ii) at step $i \leq k$ the value of the numeric fluent is computed from an expression involving the non-constant duration of a durative action, or

(iii) it is updated by a time-dependent effect expression from an action at step $i < k$,

and there is no step j at which the fluent is assigned to a non-time-dependent value, where $i < j < k$.

Definition 3.19. An expression evaluated at step k of a plan is time-dependent at k if one of its terms is a time-dependent numeric fluent at step k.

Let $\tilde{\vartheta} \subseteq \vartheta$ be the set of fluents for which there exists at least one step in the plan at which they are time-dependent. Expressions that do not depend on fluents in $\tilde{\vartheta}$ are evaluated prior

[1]If two actions are non-mutex the ϵ can actually be removed to allow both actions to commence simultaneously. POPF (Coles et al., 2010) takes this even further by only enforcing a partial ordering between mutex actions instead of a total ordering on the whole action sequence of the plan.

to the scheduling stage and excluded from the LP. Apart from potentially reducing the number of variables, this approach also helps to prune actions deemed inapplicable due to non-time-dependent numeric preconditions, thus eliminating the need to compute their LP. This approach also allows more complex non-time-dependent numeric expressions to be incorporated at the endpoints of each action, which otherwise would not be supported by the LP. One example of this would be the case where the discrete numeric effect at the start or end of a durative action is the result of a multiplication of two or more numeric fluents. Furthermore, for each step k, conditions and effects that involve fluents in $\tilde{\vartheta}$, but that are not time-dependent at k, are also excluded from the LP and are instead computed by the planner. For each $v \in \tilde{\vartheta}$, two variables v_i and v_i' are added to the LP at each step i, corresponding to the value of v before and after applying the action at step i respectively. The value of v_0 corresponds to that of the initial state, and constraints are added to enforce that $v_0 = s_0.V(v)$. Values computed by the planner for steps in which v is non-time-dependent are also enforced in the LP in the same way. The time-dependent preconditions of each action in the plan at each step i are encoded as constraints over $\tilde{\vartheta}_i$. The time-dependent invariants for a step i are also encoded in a similar fashion over $\tilde{\vartheta}_i'$ and $\tilde{\vartheta}_{i+1}$. The time-dependent instantaneous effects of each action in the plan at each step i on a variable $v \in \tilde{\vartheta}$ are encoded as constraints over the relationship between v_i and v_i'.

3.8 Summary

This chapter introduced the key concepts involved in modelling temporal problems and reasoning with concurrency. Each durative action is split into two snap actions, to allow for two durative actions to overlap in their execution and still be able to perform forward-chaining search. The characteristics of concurrency were analysed in detail, and the concept of interference between two actions was extended to durative actions. Continuous effects are defined as continuous update functions on numeric state variables, and the state transition function was combined with a state flow function to include continuous effects between discrete states. This model was extended to support timed initial literals and timed initial fluents, which can be seen as instantaneous actions without preconditions, programmed to occur at a specific time-point irrespective of the chosen plan.

This chapter follows with a representation of the temporal plan as a linear program. The numeric preconditions and effects, together with invariants, continuous effects and duration constraints are modelled as one set of linear inequalities. A linear program can then find the solution that satisfies all these constraints, determining the time-point of each happening, together with the value of each numeric state value at each corresponding discrete state. This process is incorporated within the forward-chaining search mechanism, and also acts as a further feasibility check to make sure that the partial plan constructed so far is valid. This chapter concludes with the definition of time-dependence, which identifies the variables that need to be included in the linear program, thus restricting updates on them to be linear, and those that can be excluded and precomputed by the forward-search state transition process of the planner. A non-time-dependent numeric fluent can become time-dependent at subsequent steps of the plan, and vice-versa.

4. Planning with Constants in Context

Continuous change has an important role when modelling real-world temporal problems. This was recognised in PDDL 2.1 (Fox and Long, 2003), where durative actions and continuous effects were introduced to enable such a capability (Cushing et al., 2007). Continuous effects are defined in terms of a rate of change of a numeric fluent with respect to time, as described in Section 3.3. Current state-of-the-art temporal and numeric planners, such as TM-LPSAT (Shin and Davis, 2005), COLIN (Coles et al., 2012), POPF (Coles et al., 2010) and OPTIC (Benton et al., 2012), are quite effective in dealing with linear continuous change because they reduce the problem to a linear program. One key restriction, intrinsic to the forward-chaining algorithm used by these planners, is the assumption that the rate of change of a continuous effect of a durative action is constant throughout its execution. In this chapter, a new algorithm will be introduced that removes this restriction, to support constants in context. This means that the rate of change is only assumed to be constant within a temporal context, that is between two adjacent discrete happenings. The proposed technique supports a new class of problems that current temporal numeric planners do not support, while still taking advantage of the scalability offered by linear programming.

4.1 Motivation

Continuous effects are defined in terms of a rate of change of a numeric fluent with respect to time. The PDDL semantics allow a numeric fluent to increase or decrease at a rate that can be expressed in terms of other numeric fluents combined together with simple arithmetic operators ($+$, $-$, $/$, $*$), as described in Section 2.3. The underlying assumption of current temporal planners is that these fluents have to be constant throughout the execution of the durative action. However, as explained in Section 1.1.2, it is quite possible that the rate of change of a linear continuous effect changes at discrete happenings of a plan. There are various real-world examples of this kind of behaviour, such as changes in power consumption due to switching an electrical appliance on or off, or changes in tariffs due to different peak or off-peak time bands.

COLIN and POPF support cumulative piecewise linear continuous effects. That is, linear continuous effects from multiple overlapping durative actions, each with a constant rate of change throughout its execution. The work presented in this chapter proposes a new algorithm that also supports constants in context.

4.2 Existent Approaches to Piecewise Linear Continuous Effects

While support for piecewise linear continuous effects is somewhat lacking in current state-of-the-art temporal and numeric planners, there has been significant interest in these characteristics in other related fields such as constraint satisfaction and scheduling. The following are some of the current approaches that can be used to solve a problem with these characteristics.

4.2.1 Compilation into Linear Continuous Effects

The first possibility is to translate the problem into one whose actions are restricted to linear continuous effects. If all the possible rates of change are finite and known beforehand (or can be computed as part of a pre-processing step), and it is possible to enumerate all of the permutations, an action with a piecewise linear continuous effect can be split into multiple action segments with linear continuous effects. Clip actions have to be introduced to force the action segments to act in sequence without any time gaps. Additional predicates have to be introduced as gate keepers to only allow each action segment to operate within the region where the real rate of change corresponds to that of its action segment's continuous linear effect. This enables the problem to be solvable by current temporal and numeric planners such as COLIN (Coles et al., 2012) and POPF (Coles et al., 2010).

This approach is only feasible where the possible rates of change are finite and can be determined prior to the planning stage. If an action that modifies the rate of change can be applied an arbitrary number of times during the plan, it is not possible to know all the possible values prior to the planning stage. Furthermore, while this approach is feasible for certain kinds of problems, its practicality becomes questionable with larger problem instances involving several rate changes and numerous possible durative actions with such continuous effects.

4.2.2 Constraint Satisfaction

A somewhat older but still relevant approach is an extension to CSP-based models to support piecewise linear functions over continuous domains (Trinquart and Ghallab, 2001). In this approach the representation used by the CSP-based partial-order causal link (POCL) planner IxTeT was extended to model this kind of behaviour. IxTeT uses its own formalism to model tasks (the planning operators corresponding to PDDL actions), which are combined together to form *chronicles* (a partial plan). Temporal constraints are introduced over these tasks within a chronicle. Furthermore, logical constraints are also enforced over the state variables. Events indicate instances where there is a discrete state transition, which includes preconditions and effects.

The addition of an extra *change* temporal construct introduced the concept of continuous linear change. This is a constraint of the form $change(p : (v, v'), (t, t'))$, which specifies that the value of attribute p changes linearly from v to v' during the interval t to t', expressing the constraint $v' - v = a * (t' - t)$. IxTeT performs a least commitment search (Weld, 1994) in the space of partial plans to find a chronicle which satisfies all the constraints, including the temporal and numeric ones. In order to handle this additional complexity its CSP-solver was extended to support the specified constraints. This was done by decomposing the domain of

a continuous numeric variable into disjoint intervals, corresponding to each constraint. The propagation algorithm then tries to bound the domain of that variable to these intervals, thus reducing the continuous nature of the problem to a discrete and finite one that can be solved using CSP techniques.

While this approach is very specific to the problem representation used by IxTeT, the general idea is in line with that used in modern state-of-the-art planners. As described in Section 3.6, temporal and numeric constraints are modelled in a very similar way. The only difference is the solver used, in which case modern temporal and numeric planners typically use linear programming.

4.2.3 Scheduling with Linear Resources

The same characteristics were also studied in the realm of scheduling. A Linear Resource Temporal Network (LRTN) consists of activities that consume or produce resources, restricted to linear functions (Frank and Morris, 2007). It is an extension to Resource Temporal Network (RTN) (Laborie, 2003), which in itself consists of a Simple Temporal Network (STN) (Dechter et al., 1991) combined with instantaneous effects on resources at each discrete time-point. An LRTN also allows gradual production or consumption of resources in between discrete time-points of the RTN. Activities in an LRTN consist of a start, end and duration, and their effect on resources can be linear with respect to the duration. Resources have upper and lower bounds, which can be piecewise linear functions.

LRTNs are assumed to have a single resource (Frank and Morris, 2007) and the activities are known beforehand. In the case of temporal and numeric planning, multiple resources can be consumed or produced concurrently, and the choice of which actions to apply also needs to be determined by the planner. Nevertheless, the LRTN is solved through a linear program which consists of variables representing the resource capacities. This is also similar to the approach used in modern temporal and numeric planners, which solve the STN corresponding to the temporal constraints of a plan, in conjunction with the numeric constraints reflecting the action preconditions, effects, and invariant conditions, as described in Section 3.6. A fundamental difference however is that in forward-chaining planners such as COLIN this process takes place for each candidate state to verify its validity and temporal feasibility.

4.2.4 Time Discretisation

Hybrid planners such as UPMurphi (Della Penna et al., 2009) take a different approach towards solving problems with continuous effects. They aim to support a wider range of functions, including non-linear continuous functions, as both effects and constraints. In order to support such a wide range of functions the time-line is discretised into discrete time-steps, and the search process is then performed across these time-steps.

While this approach has a significant advantage of supporting all the possible functions that can be described using the respective modelling language (in this case PDDL+), it has a significant drawback when it comes to scalability. A trade-off has to be made between granularity and accuracy, and the user of the planner has to select the size of these time-steps. Selecting a small time-step size often leads the planner to get lost in search until it runs out of

memory. On the other hand a larger time-step size limits the time points at which actions can occur and can lead to an invalid result that violates the numeric constraints. In order to use this planner effectively a very short planning horizon has to be used.

4.3 Planning with Piecewise Linear Continuous Effects

In the work proposed in this chapter we build on existing ideas which make use of linear programming, as described in Section 3.6. This has the advantage of polynomial scalability, accurate numeric computation, and supports plans of arbitrary makespan. We improve the algorithm used by COLIN (Coles et al., 2012) to support what we refer to as constants in context. A temporal context refers to the interval between two adjacent discrete happenings in the plan, as defined in Definition 3.14. While in COLIN, a continuous effect of a durative action is assumed to be constant throughout its execution, with constants in context we allow another discrete action (or timed initial fluent) to change the rate of change of the durative action's continuous effect. With this enhancement, durative actions can effectively have piecewise linear continuous effects.

The COLIN algorithm is already capable of combining concurrent continuous effects on the same variable, v, by sequentially accumulating the increase or decrease of its rate of change with each start or end action that has a continuous effect on v (Coles et al., 2012). Continuous effects on a numeric fluent $v \in \tilde{\vartheta}$ at step i are represented with constraints on the relationship between v_i' and v_{i+1}, where v_i' corresponds to the value of the variable at step i after applying discrete effects, while v_{i+1} corresponds to the value of the variable at step $i+1$ before applying discrete effects. Figure 4.1 illustrates this process, with two durative actions, a, and b, both having a continuous effect on a variable v. One global rate of change, dv, is maintained for v, which is updated at the start and end of each action that has a continuous effect on v.

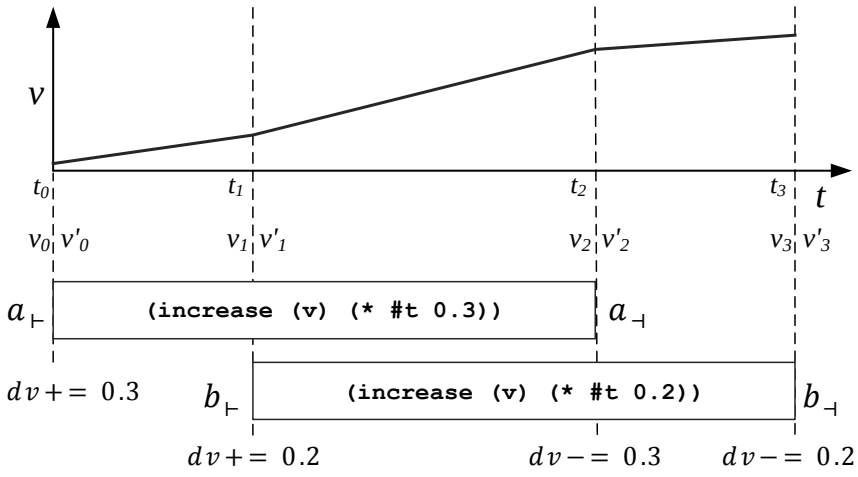


Figure 4.1: Tracking Linear Continuous Effects in COLIN

In COLIN, the rate of change of an action's continuous effect is assumed to be constant throughout the execution of the durative action. This approach does not support cases where an action modifies the rate of change of a continuous effect of another action that has already

started. Figure 4.2 illustrates one such example, where action a performs a linear increase on variable v at rate y, and the start snap action $b_\vdash$ updates y, affecting the rate of change of the rest of the continuous effect of a.

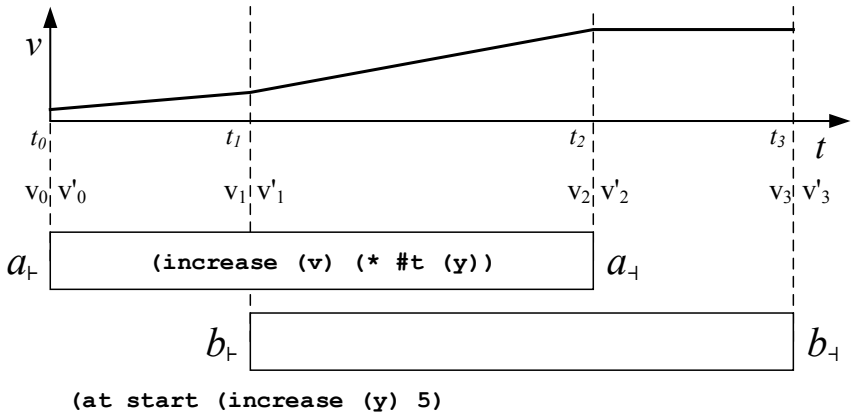


Figure 4.2: Updating the rate of change of a continuous effect.

Rather than accumulating the rate of change sequentially, we compute a fresh value for the cumulative rate of change, $\mathrm{d}v_i$, for each step i. Let $conc(i)$ be the list of all durative actions known to be running concurrently within the context of step i. Note that there is a subtle difference between $conc(i)$ and $exec(t)$ defined earlier in Definition 3.5. $exec(t)$ refers to actions running at a specific time point t, while $conc(i)$ refers to the actions planned to run concurrently together from step i up till step $i + 1$ (even if the actual times of steps i and $i + 1$ are not known yet). Let $ceffs_i$ represent the multiset of continuous effects active at step i up till step $i + 1$. This can be derived from Equation 3.3 defined earlier as follows.

$$ceffs_i = \biguplus_{\dot{o}_{dur} \in conc(i)} \{\langle v, expr_\tau \rangle \to 1 | \langle v, expr_\tau \rangle \in contEff(\dot{o}_{dur})\} \tag{4.1}$$

From Equation 3.6 we can now derive the rate of change for v_i. Each continuous effect expression, $expr_\tau$, needs to be evaluated in terms of the state, s_i, for each step i, and thus can change the trajectory of v.

$$\frac{\mathrm{d}v_i}{\mathrm{d}t} = \sum_{\langle v, expr_\tau \rangle \to n \in ceffs_i} n \cdot expr_\tau(s_i) \tag{4.2}$$

The actual change of v, starting from step i up till step $i + 1$, denoted Δv_i, depends on the time chosen for each of the steps. Thus, the value of v for the subsequent step can then computed as shown in Equation 4.4.

$$\Delta v_i = \frac{\mathrm{d}v_i}{\mathrm{d}t}(ts(i+1) - ts(i)) \tag{4.3}$$

$$v_{i+1} = v_i' + \Delta v_i \tag{4.4}$$

This approach allows for discrete updates to the rate of change of a linear continuous effect to take place, with a different value for Δv in each context. Figure 4.3 illustrates this process. Action a is increasing v at a rate y. Action b is also increasing v at a rate z, but before that it

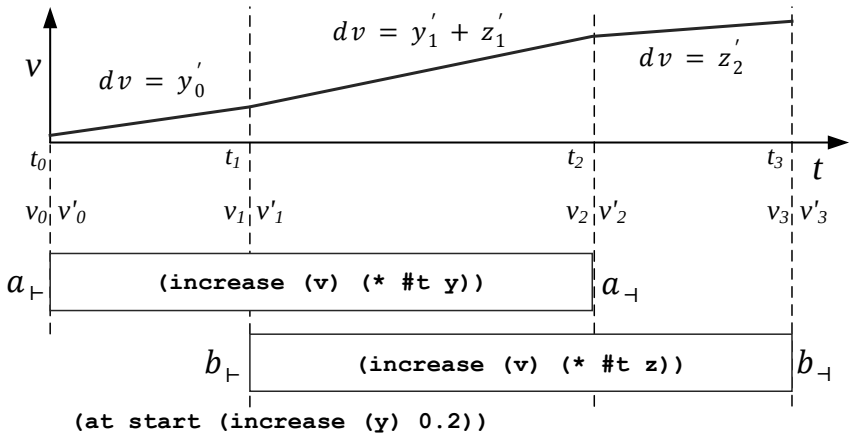


Figure 4.3: Tracking piecewise linear continuous effects.

performs a discrete increase to y, updating the continuous effect of action a. Calculating the rate of change of v from the new state obtained from applying the state progression at each happening enables the correct evaluation of dv for each context in the plan.

4.4 Forward Search with Durative Actions and Timed Happenings

Each durative action a_{dur} is split into two snap actions: $a_{\vdash}$, representing the conditions and effects when the durative action starts, and $a_{\dashv}$, representing the conditions and effects when the durative action ends. The set of operators of P_τ can thus be defined as $O = O_{inst} \cup \{a_{\vdash}, a_{\dashv} \mid a_{dur} \in O_{dur}\}$. Refer to Section 3.1 for more details.

A temporal state is defined as $s = \langle F, V, \pi, Q, C, T \rangle$, where:

- F is the set of atomic propositions holding in the state, with $F \subseteq \rho$.
- V maps the numeric variables to their respective value, with $V : \vartheta \to \mathbb{R}$
- π is the plan, a sequence of (unscheduled) actions $(a_1, a_2, ..., a_n)$ where $a_i \in O$, to reach the state s from s_0.
- Q is a set of executing durative action instances that have commenced, but not yet terminated, together with the step index at which they were started in the plan π. Each element in Q takes the form $\langle a_{\vdash}, i \rangle$, where $i \in [1..n]$.
- C is a set of temporal constraints of the form $lb \leq (ts(j) - ts(i)) \leq ub$ for step indices $\{i, j\} \subseteq \{0, ..., n\}$ in plan π, where lb is the lower-bound and ub is the upper-bound of the temporal interval between steps i and j, and $ts : \mathbb{N} \to \mathbb{R}_{\geq 0}$ determines the time at which a step is to be executed.
- T is a queue of timed happenings, $\langle t_1, t_2, ..., t_k \rangle$, for which the respective steps have not been added to the plan yet, ordered by earliest first.

Figure 4.4 illustrates this representation, consisting of a simple temporal network over the plan of the preceding steps prior to the current state. The actions a_1 and a_3 started and ended in the plan, while action a_2, commenced at step 2 but has not yet terminated.

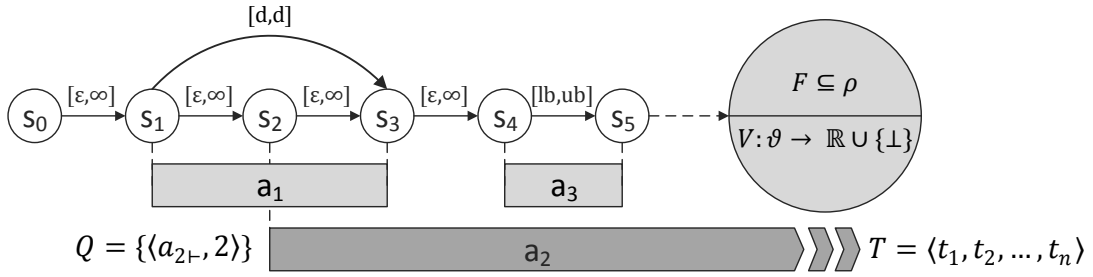


Figure 4.4: Temporal state information.

Whenever an action, $a \in O$, is applied and appended to the plan π to obtain a new state s', the propositions F and numeric fluents V are updated according to the effects, $eff(a)$. Furthermore, if a is a start or end snap action of a durative action a_{dur}, Q and C need to be updated accordingly. If $a = a_{\vdash}$, Q is updated with an entry containing the snap action and the index of the step, $Q' = Q \cup \{\langle a_{\vdash}, |\pi|\rangle\}$. A constraint is also added to C to enforce that $\epsilon \leq ts(|\pi|) - ts(|\pi| - 1)$, where ϵ is a very small non-zero amount enforcing a total ordering between actions (Coles et al., 2012). If $a = a_{\dashv}$, new constraints of the form $lb \leq (ts(|\pi|) - ts(i)) \leq ub$ are added to C, given that the corresponding entry for the start snap action, $\langle a_{\vdash}, i\rangle$, is in Q. These constraints reflect the duration condition, $durCond(a_{dur})$, of the durative action. The entry $\langle a_{\vdash}, i\rangle$ is then removed from Q of the new state s'.

Similarly, whenever one of the pending timed happenings, t, is removed from the queue, T, the corresponding cumulative timed action, o_t, is applied to obtain a new state, s', and the corresponding step is added to the plan, π. A constraint is also added to $s'.C$ to enforce that $ts(|s'.\pi|) = t$.

For a state s to be valid, the solution to the simple temporal network of its plan, π, taking into consideration the temporal constraints, C, must have a makespan which is less than or equal to that of the first (earliest) pending timed happening in T. That is, $ts(|\pi|) \leq t$, where $T = t : ts$. This ensures that any TILs and TIFs that are scheduled to happen during the plan's execution are included.

The invariant conditions that must hold throughout the duration of a state s consist of the invariant conditions of all the actions for which an entry exists in Q, as defined in Equation 4.5, where $Dom(Q) = \{a_{\vdash} | \langle a_{\vdash}, i\rangle \in Q\}$ and a_{dur} is the durative action commenced by $a_{\vdash}$.

$$stateInv(s) = \bigcup_{a_{\vdash} \in Dom(Q)} inv(a_{dur}) \tag{4.5}$$

According to PDDL 2.1 semantics, continuous effects take place while in a state, as time progresses from one step in the plan to the next. Variables affected by continuous change are time-dependent, since their value does not depend solely on which actions are chosen, but also on their duration. In our planning framework, continuous effects on variables can either increase or decrease the value of a numeric fluent by a constant rate throughout a state's duration

(context). It can thus be formally defined as a tuple $\langle v, \mathrm{d}v \rangle$, where $v \in \vartheta$ and $\mathrm{d}v$ is the rate of linear change. The list of continuous effects taking place between a state s_i and s_{i+1}, $ceffs_i$, corresponds to those durative actions whose start snap action is in $Dom(s_i.Q)$.

4.5 A Planning Algorithm for Constants in Context

The `applyAction` function shown in Algorithm 1 is responsible for temporal state progression. This function applies the necessary constraints and updates the data stored in the temporal state, depending on whether the action represents:

- TILs or TIFs at a specific happening t,
- an end snap action of a durative action already running in the plan (there is a record of the start snap action in Q),
- a start snap action of a new durative action, or
- an instantaneous action.

The `schedule` function shown in Algorithm 2 attempts to find a valid schedule for the plan. It first runs a linear program on the temporal constraints, through the `lpSchedule` function (as described in Section 3.6), and then checks that it is still valid with respect to the pending timed happenings.

Algorithm 3 is a recursive algorithm that keeps track of an *open list* of states, and avoids re-exploring *seen* states (those that have already been expanded). The base cases for this algorithm are when either there are no states left to evaluate in the open list, or when the goal condition is achievable with the plan in the state that is being explored. Since the goal could be a numeric goal that is time dependent, a dummy action a_G is created through the `goalAction` function, where $pre(a_G) = G$ and $eff(a_G) = \langle \emptyset, \emptyset, \emptyset \rangle$. The `nt` function extracts a conjunction expression consisting of propositional and non-time-dependent numeric fluents. This is used to evaluate the initial applicability of an action prior to running the scheduling step (which involves a linear program). Prior to exploring further states, if the non-time-dependent parts of the goal expression, G, are satisfied by s, a_G is appended to the plan of s, and an attempt is made to schedule the plan. If this is successful, a solution has been found. This handles cases where the plan of the current temporal state can actually achieve the goal conditions if subjected to the right schedule.

Otherwise, the actions whose non-time-dependent preconditions are satisfied by s are applied, obtaining a set of new (unscheduled) states, U, excluding any of those which are in the set `seen`. The action, o_t, corresponding to the next timed happening, t (if any), is also applied to the state, s, obtaining a singleton set of states, S_t. All the states in U, together with the state in the singleton set S_t, are scheduled such that those which cannot have a valid schedule are filtered out, obtaining the set of valid possible successor states, S_v.

The function is tail recursive. The new set of unexpanded states is combined with the previous tail of the open list, sx, using the `combine` function. The state that has just been expanded, s, is added to the set `seen`. The new open list and the set of `seen` states are passed

Algorithm 1: Apply a Simple Action

Function `applyAction(s, o)`

Data: The temporal state, s, to which the action o is being applied. O_{dur} is the global list of durative actions for the planning problem.

Result: A new temporal state s'.

if $s.T = \langle\rangle$ **then**
  $t \leftarrow 0$
else
  $t : tx \leftarrow s.T$
$s'.F \leftarrow (s.F \setminus \mathit{eff}^{-}(o)) \cup \mathit{eff}^{+}(o))$
$s'.V \leftarrow \mathit{eff}^{\vartheta}(o)s$
$s'.\pi \leftarrow s.\pi \cdot o$
if $o = o_t$ **then**
  $s'.T = tx$
  $s'.C = s.C \cup \{t \leq ts(o) \leq t\}$
  $s'.Q = s.Q$
else
  $s'.T \leftarrow s.T$
  if $o \in \{o_{\dashv} \mid \langle o_{\vdash}, t_s\rangle \in s.Q\}$ **then**
    $\dot{o}_{dur} = \mathit{durInst}(o)$
    $s'.C \leftarrow s.C \cup \mathit{durCond}(\dot{o}_{dur}) \cup \{\epsilon \leq ts(|s.\pi| + 1) - ts(|s.\pi|)\}$
    $s'.Q \leftarrow s.Q \setminus \{\langle \mathit{start}(\dot{o}_{dur}), t_s\rangle \mid \langle \mathit{start}(\dot{o}_{dur}), t_s\rangle \in s.Q\}$
  else
    $s'.C \leftarrow s.C \cup \{\epsilon \leq ts(|s.\pi| + 1) - ts(|s.\pi|)\}$
    if $o \in \{o_{\vdash} \mid o_{dur} \in O_{dur}\}$ **then**
      $s'.Q \leftarrow s.Q \cup \{o, (|s.\pi| + 1)\}$
    else
      $s'.Q \leftarrow s.Q$
return s'

as arguments to the next recursive call. Note that this search algorithm can be used in both breadth-first and best-first mode. The only difference would be that in the case of breadth-first search the open list is a queue data structure, while in best-first search the open list is a priority queue (heap), ordered by the state's value according to a heuristic evaluation function. The heuristic used in this work is based on the length of the relaxed plan extracted from the TRPG of the state, as described in Section 4.6. It is also possible to use Enforced Hill-Climbing (EHC), and remove all the queued states from the open list whenever a state with the current best heuristic value is found. Section 4.7 describes a variation of EHC adapted to perform better when solving planning problems that feature numeric and temporal constraints.

4.6 Adapting the Delete Relaxation Heuristic

The Temporal Relaxed Planning Graph (TRPG) (Coles et al., 2012) adapts the Metric-FF delete relaxation heuristic (Hoffmann, 2003) which is based on the Relaxed Planning Graph (Hoffmann and Nebel, 2001) to carry temporal information with each action layer. The actions

Algorithm 2: Schedule a Plan

Function `schedule(`s`)`

Data: The temporal state, s, for which a valid scheduled temporal plan is required.

Result: A pair containing a valid scheduled temporal plan and the state with the corresponding values for the time-dependent numeric fluents, or Failure

$\langle \pi_\tau, s' \rangle \leftarrow$ `lpSchedule`$(s.\pi)$

if $(\pi_\tau \neq \text{Failure})$ **and** $(s.T = \langle\rangle)$ **then**

  return $\langle \pi_\tau, s' \rangle$

else if $(\pi_\tau \neq \text{Failure})$ **then**

  $t : tx \leftarrow s.T$

  if $(ts(|\pi_\tau|) \leq t)$ **then**

    return $\langle \pi_\tau, s' \rangle$

return Failure

Algorithm 3: Search for a Plan

Function `searchForPlan(`open, seen, O, G`)`

Data: The list of open states that still need to be explored, the set of seen states, the set of possible operators O, and the goal condition G.

Result: A valid scheduled temporal plan to achieve G, or Failure

if open $= \langle\rangle$ **then**

  return Failure

$s : sx \leftarrow$ open

if $s \notin$ seen **then**

  if $(s.Q = \emptyset)$ **and** $(s \models \texttt{nt}(G))$ **then**

    $a_G \leftarrow$ `goalAction`(G)

    $\langle \pi_\tau, s_G \rangle \leftarrow$ `schedule`$(s.\pi \cdot a_G)$

    if $\pi_\tau \neq$ Failure **then**

      return π_τ

  $U \leftarrow \{\texttt{applyAction}(s, o) \mid o \in O \wedge s \models \texttt{nt}(pre(o))\} \setminus$ seen

  if $s.T = \langle\rangle$ **then**

    $S_t \leftarrow \emptyset$

  else

    $t : tx \leftarrow s.T$

    $S_t \leftarrow \{\texttt{applyAction}(s, o_t)\}$

  $S_v \leftarrow \{s_v \mid s_v \in U \cup S_t \wedge \texttt{schedule}(s_v) \neq \text{Failure}\}$

  `searchForPlan(combine(`sx, S_v`), seen` $\cup \{s\}$`)`

else

  `searchForPlan(`sx, seen`)`

in each layer do not just correspond to those whose preconditions are satisfied by the previous proposition layer, but also to the set of actions whose earliest time of application corresponds to the time-stamp of their action layer.

The structure of the TRPG is similar to the one constructed by Metric-FF. It carries additional information associated with each layer, this being the earliest time-stamp at which the layer can optimistically take place in a plan. When building the TRPG, the applicability of a simple

action considers its preconditions (together with invariants for start snap actions) and also any temporal constraints. TILs and TIFs have to appear in an action layer where the time-stamp matches the one defined by the TIL or TIF. The time-stamp of a layer where an end snap action appears also has to satisfy its temporal constraints in relation to its corresponding start snap action. This approach is analogous to that used in CRIKEY3 (Coles et al., 2008), COLIN (Coles et al., 2012) and POPF (Coles et al., 2010).

An enhanced version of the TRPG used by these planners is proposed in this work, referred to as TRPGne (Temporal Relaxed Planning Graph - *numeric-enhanced*), which is designed to support richer numeric causality. The first difference between the TRPGne and the TRPG is that in TRPGne the applicable actions consider cases where their numeric effect depends on some variable that is updated by another action. Such an update could affect which bounds of a numeric variable are updated in subsequent layers. For example, if an action, a, has an effect `(increase (y) (x))` and the bounds of x are $\langle 0, 0 \rangle$, the bounds of y will not be affected (assuming no other action updates y). However, if another action, b, has an effect `(increase (x) (1))`, subsequent layers where a is applicable (which will always be the case due to the monotonically increasing facts property of the RPG) have to also take this into consideration, and increase the upper bound of y accordingly. Similarly, if another action, c, has an effect `(decrease (x) (1))`, the effect of action a in subsequent layers would be that the lower bound of y would also start to decrease monotonically. This becomes even more important in continuous linear effects, where the polarity of the gradient is inverted by a discrete action executed concurrently.

The second difference, which follows from the first, is in the way the relaxed plan is extracted from the TRPGne. Apart from following the procedure of the metric RPG as defined in Metric-FF (Hoffmann, 2003), where each time an action is added to the relaxed plan its preconditions are added as intermediate goals that need to be achieved, we also need to consider *implicit intermediate goals* that need to be achieved in order to achieve some numeric goal. More specifically, these implicit intermediate goals are numeric conditions on the rvalue of discrete or continuous numeric updates. For example, if the initial state of a problem defines the value of a numeric fluent y to be 0, the goal condition of the task is to have $y > 0$, and the only action that increases y is a durative action whose continuous effect is `(increase (y) (#t (x)))`, then the fact that at some point the TRPGne satisfies the goal condition implies that at some layer the condition $x > 0$ was satisfied. In this case $x > 0$ becomes an implicit intermediate goal for the preceding layers, and the earliest action that achieves it is included in the relaxed plan (unless it was already achieved in the initial fact layer). Such a layer could occur after the one featuring the start snap action of the durative action, interfering with its continuous effect.

Another minor difference is that the TRPGne supports negative action preconditions for domains that indicate this through the `:negative-preconditions` requirement. This simplifies domains where a boolean proposition indicates whether an activity took place or not. This is especially useful in temporal planning, where actions need to be restricted from executing more than once concurrently, or more than once throughout the plan. Note that this can still be achieved easily with COLIN's TRPG, but it requires extra *mirror predicates* that indicate the opposite of their counterparts. For example, if a durative action, a, is associated with a

predicate `(running a)` to indicate that it is executing, a mirror predicate `(not-running a)`, would indicate the opposite. The TRPGne handles this implicitly in its internal representation.

4.6.1 Building the Numeric-Enhanced TRPG (TRPGne)

The TRPGne is a layered graph, consisting of alternating fact and action layers. Each fact layer, fl_n, is associated with a time-stamp $ts(fl_n)$, where $ts(fl_0) = 0$ and $ts(fl_{n+1}) > ts(fl_n)$. A fact layer, fl_n, corresponds to a relaxed state, which is a tuple $\langle F, \bar{F}, V_b \rangle$ where:

- F is the set of atomic propositions holding in the relaxed state, with $F \subseteq \rho$.
- $\bar{F}$ is the set of atomic propositions that were negated in previous action layers, with $\bar{F} \subseteq \rho$.
- V_b maps the numeric variables to their respective bounds $V_b : \vartheta \rightarrow \langle \mathbb{R}, \mathbb{R} \rangle \cup \langle \perp, \perp \rangle$.

The first fact layer of the TRPGne, fl_0, corresponds to the relaxed state obtained from assigning the state variables of the current temporal state, s.

$$fl_0.F = s.F \tag{4.6}$$

$$fl_0.\bar{F} = \emptyset \tag{4.7}$$

$$fl_0.v = \begin{cases} \langle \min(s.v), \max(s.v) \rangle & \text{if } v \in \tilde{\vartheta} \\ \langle s.v, s.v \rangle & \text{otherwise} \end{cases} \tag{4.8}$$

$\tilde{\vartheta}$ is the set of time-dependent fluents, and $\max(s.v)$ and $\min(s.v)$ correspond to the maximum and minimum possible values of v respectively for the plan of temporal state s. These values can be found by solving a linear program in the same way as described in Section 3.6, while maximising or minimising the value of v at the last happening (instead of minimising the time-point of the last happening).

An action layer, al_n, corresponds to the set of applicable actions in the relaxed state of fact layer fl_n. A relaxed state, r, satisfies the pre-conditions of an action, o, denoted $r \models pre(o)$, if:

- $pre^+(o)$, the set of all the atomic propositions required to be true in $pre(o)$, is in $r.F$, that is $pre^+(o) \subseteq r.F$,
- $pre^-(o)$, the set of all the atomic propositions required to be false in $pre(o)$, are not in $r.F$ or else are in $r.\bar{F}$, that is $(pre^-(o) \cap r.F) \subseteq r.\bar{F}$, and
- at least one permutation of the minimum and maximum values of $r.V$ satisfies all the numeric conditions in $pre^{\vartheta}(o)$.

An action, o, is applicable in the TRPGne at layer al_n if $fl_n \models pre(o)$ and $ts(fl_n) \leq earliest(o)$. For start snap actions and instantaneous actions $earliest(o) = 0$. For end snap actions, $earliest(o)$ is equal to the minimum time that has to elapse from the corresponding start snap action, appearing earlier on in the plan or in a previous action layer. Any start snap action must also be checked for the invariant conditions of its respective durative action, in the relaxed

state obtained after applying the action layer. That is $\mathit{fl}_n \models \mathit{pre}(o_\vdash)$ and $\mathit{fl}_{n+1} \models \mathit{inv}(o_{dur})$, where $o_\vdash$ is the start snap action of durative action o_{dur}.

The subsequent fact layer, fl_{n+1}, consists of all the propositional facts in the previous layer, together with any new facts affirmed or negated by actions in al_n. That is, $\mathit{fl}_{n+1}.F = \mathit{fl}_n.F \cup \bigcup\{\mathit{eff}^+(o)|o \in al_n\}$ and $\mathit{fl}_n.\bar{F} = \mathit{fl}_n.\bar{F} \cup \bigcup\{\mathit{eff}^-(o)|o \in al_n\}$. The minimum and maximum bounds of numeric variables are also decreased or increased respectively for each applicable action. Thus, the minimum bound of each numeric variable decreases monotonically, while its maximum bound increases monotonically, with each subsequent layer.

4.6.2 Extracting the Relaxed Plan from the TRPGne

The mechanism of extracting a plan from the TRPGne is similar to that used in Metric-FF (Hoffmann, 2003). However, apart from considering explicit achievers of action preconditions, it also considers achievers of implicit intermediate conditions necessary for discrete and numeric continuous effects to satisfy an explicit goal.

Algorithm 4: Analyse the TRPGne for the actions required for the relaxed plan.

Function `analyse` *(*trpgne*, n, G, A)*

Data: A numeric-enhanced temporal relaxed planning graph, trpgne, the layer, n, currently being analysed, the set of conjunctive atomic goal conditions G, and $A : \mathbb{N}_{>0} \nrightarrow O^n$ mapping each layer (by index) to the set of required actions for the relaxed plan.

Result: The partial function $\mathbb{N}_{>0} \nrightarrow O^n$ that maps layer indices to the set of actions selected from that layer for the relaxed plan.

if $n > 0$ **then**

- $G_n \leftarrow \{g | g \in G \wedge \mathsf{trpgne}.fl_n \models g \wedge \mathsf{trpgne}.fl_{n-1} \not\models g\}$
- $a_n \leftarrow \{\texttt{selectAchiever}(\mathsf{trpgne}, n, g) | g \in G_n\}$
- $G_{pre} \leftarrow \{\mathit{pre}(a) \cup \mathit{inv}(a) | a \in a_n\}$
- $G_\vartheta \leftarrow \texttt{numericPrec}(G_n)$
- $\texttt{analyse}(\mathsf{trpgne}, n-1, (G \setminus G_n) \cup G_{pre} \cup G_\vartheta, A \cup \{n \rightarrow a_n\})$

else

- **return** A

Algorithm 4 performs the analysis of the TRPGne to find the actions required to construct the relaxed plan. For simplicity, the goal condition, G, and the preconditions, $\mathit{pre}(a)$ and invariants, $\mathit{inv}(a)$, of an action, a, are assumed to be a set of conjunctive atomic propositional or numeric facts. The `selectAchiever` function selects one action from layer n that achieves an atomic goal g. Similarly, the `numericPrec` function returns the numeric conditions required for any numeric goals, G_n, achieved in layer n. This is done by taking the rvalue of the effect that achieved the respective goal, and constructing a new precondition which needs to be achieved by an action in a prior layer. For example, if the goal condition `(x > 5)` was achieved by the numeric effect `(increase (x) (y))` then a new implicit condition is added to G_ϑ stating that `(y > 0)`. This way, any action that changes y to a positive value will be included in the relaxed plan. Given that in the proposed algorithm we support constants in context, this mechanism needs to take place for both discrete and continuous numeric effects.

The returned data structure maps the layer indices of the TRPGne to the set of actions selected in each respective layer for the relaxed plan. As in FF (Hoffmann and Nebel, 2001), the actions in the first layer are considered helpful actions, and are used in Enforced Hill-Climbing, pruning out unhelpful actions from its search space.

4.7 Enforced Hill-Climbing with Ascent Backtracking

Enforced Hill-Climbing (EHC) has proven to be a very good search strategy when combined with a delete-relaxation heuristic in classical planning (Hoffmann and Nebel, 2001), and to some degree in numeric planning (Hoffmann, 2003). COLIN (Coles et al., 2012) and POPF (Coles et al., 2010) also use EHC in temporal planning since the TRPG is based on the RPG proposed in Metric-FF (Hoffmann, 2003).

EHC suffers from an inherent weakness that early commitment decisions could lead to dead ends (Hoffmann, 2005). This is especially true in planning problems that involve temporal or numeric constraints. Since the FF heuristic uses delete-relaxation, it will eagerly guide the search to choose actions that seemingly take the state closer to the goal, ignoring the fact that such an early commitment could lead to a violation of numeric or temporal constraints. Since EHC is a local-search technique, it is not complete, and most planners resort to a fall-back search mechanism such as Weighted A* (Pohl, 1970) when EHC fails.

For the kinds of planning problems considered in this work, which involve rich numeric and temporal constraints, such as deadlines, it was observed that EHC often fails to yield a plan due to a wrong hill-climb at some point in the search. Rather than falling back to Weighted A* right away, a more forgiving EHC variant is proposed.

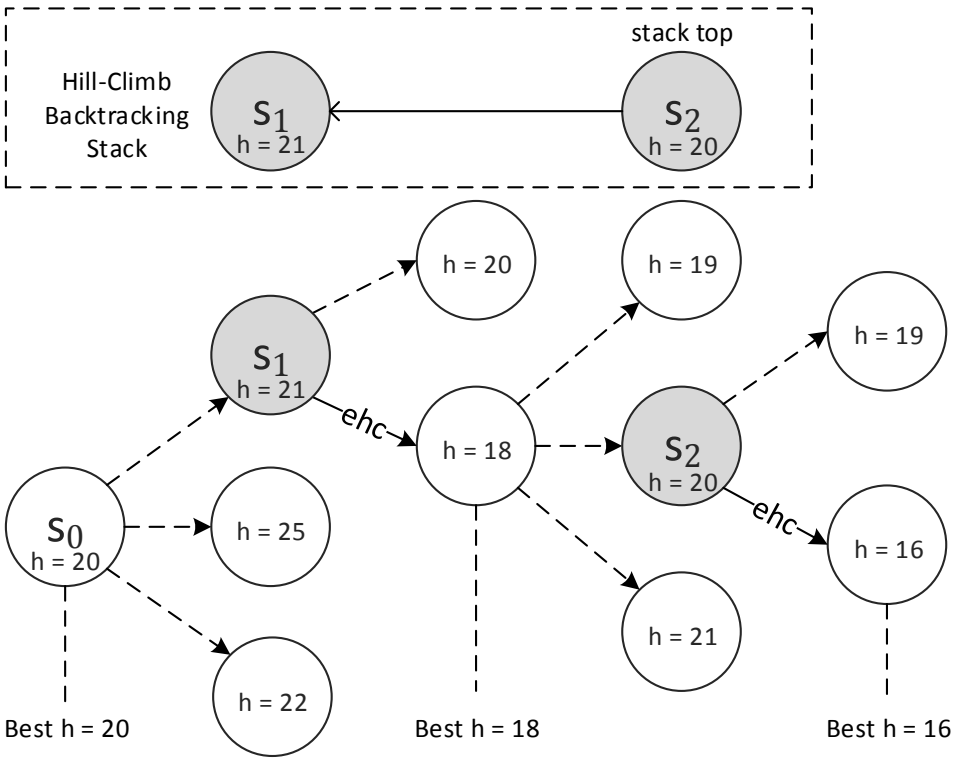


Figure 4.5: Enforced Hill-Climbing with Ascent Backtracking

Algorithm 5: Enforced Hill-Climbing with Ascent Backtracking

Function `ehcAbSearch` (open, seen, climbs, H, O, G)

Data: A queue of open states to be explored, the set of seen states, the stack of climbs corresponding to states from which a hill-climb was performed, the best heuristic value seen so far H, the set of possible operators O, and the goal condition G.

Result: A valid scheduled temporal plan to achieve G, or Failure

if open = $\langle\rangle$ **then**
  return `abSearch` (climbs, seen, O, G);

$s : sx \leftarrow$ open

if $(s.Q = \emptyset)$ **and** $(s \models \texttt{nt}(G))$ **then**
  $a_G \leftarrow \texttt{goalAction}(G)$
  $\langle \pi_\tau, s_G \rangle \leftarrow \texttt{schedule}(s.\pi \cdot a_G)$
  if $\pi_\tau \neq$ Failure **then**
    return π_τ

$O_h \leftarrow \texttt{helpfulActions}(s)$

$U \leftarrow \{\texttt{applyAction}(s, o) \mid o \in O_h \wedge s \models \texttt{nt}(pre(o))\} \setminus$ seen

if $s.T = \langle\rangle$ **then**
  $S_t \leftarrow \emptyset$

else
  $t : tx \leftarrow s.T$
  $S_t \leftarrow \{\texttt{applyAction}(s, o_t)\}$

$S_v \leftarrow \{s_v \mid s_v \in U \cup S_t \wedge \texttt{schedule}(s_v) \neq$ Failure$\}$

$S_h \leftarrow \{s_h \mid s_h \in S_v \wedge \texttt{heuristic}(s_h) < H\}$

if $S_h \neq \emptyset$ **then**
  $s_h : sx_h \leftarrow \text{seq}[S_h]$
  $\texttt{ehcAbSearch}(\langle s_h \rangle, \text{seen} \cup \{s\}, s : \text{climbs}, \texttt{heuristic}(s_h), O, G)$

else
  $\texttt{ehcAbSearch}(\texttt{combine}(sx, S_v), \text{seen} \cup \{s\}, \text{climbs}, H, O, G)$

Algorithm 6: Ascent Backtracking Algorithm

Function `abSearch` (climbs, seen, O, G)

Data: The stack of climbs corresponding to states from which a hill-climb was performed, a queue of open states to be explored, the set of possible operators O, and the goal condition G.

Result: A valid scheduled temporal plan to achieve G, or Failure

if climbs = $\langle\rangle$ **then**
  return Failure;

$s : sx \leftarrow$ climbs

$\pi_\tau \leftarrow \texttt{searchForPlanHA}(s, \text{seen}, O, G)$

if $\pi_\tau \neq$ Failure **then**
  return π_τ

else
  return $\texttt{abSearch}(sx, \text{seen} \cup s)$

Algorithm 7: Search for a Plan using Helpful Actions

```
Function searchForPlanHA(open, seen, O, G)
    Data: The list of open states that still need to be explored, the set of seen states, the
          set of possible operators O, and the goal condition G.
    Result: A valid scheduled temporal plan to achieve G, or Failure
    if open = ⟨⟩ then
        return Failure
    s : sx ← open
    if s ∉ seen then
        if (s.Q = ∅) and (s ⊨ nt(G)) then
            a_G ← goalAction(G)
            ⟨π_τ, s_G⟩ ← schedule(s.π · a_G)
            if π_τ ≠ Failure then
                return π_τ
        O_h ← helpfulActions(s)
        U ← {applyAction(s, o) | o ∈ O_h ∧ s ⊨ nt(pre(o))} \ seen
        if s.T = ⟨⟩ then
            S_t ← ∅
        else
            t : tx ← s.T
            S_t ← {applyAction(s, o_t)}
        S_v ← {s_v | s_v ∈ U ∪ S_t ∧ schedule(s_v) ≠ Failure}
        searchForPlanHA(combine(sx, S_v), seen ∪ {s})
    else
        searchForPlanHA(sx, seen)
```

The main intuition behind Enforced Hill-Climbing with Ascent Backtracking (EHC-ab), is that when EHC ends up in a dead end, it backtracks to the state prior to performing the last hill-climb. Maintaining a stack of these states requires negligible extra memory, and gives the advantage that a wrong hill-climb can be re-evaluated. At this point all the helpful actions are considered. If this also fails, the next (earlier) hill-climb is popped off the stack to reverse the previous commitment, and the process is repeated tail-recursively. Figure 4.5 illustrates this algorithm. Arrows marked *ehc* indicate that enforced hill-climbing took place, with the originating state pushed on the hill-climb backtracking stack.

While this approach is also not complete, it helps EHC recover from instances where the heuristic happens to provide improper guidance. EHC-ab is shown in Algorithms 5 to 7. The `helpfulActions` function computes the TRPGne heuristic described in Section 4.6 and returns the set of helpful actions from a state.

4.8 Computational Characteristics

The technique proposed above combines heuristic forward search and linear programming. Heuristic forward search is used to explore the possible action sequences while a linear program verifies the temporal characteristics of the plan to reach each state.

The heuristic proposed in this algorithm is an enhanced version of the temporal relaxed planning graph, which is based on the delete relaxation technique. This in itself is polynomial on the number of input actions (Hoffmann and Nebel, 2001; Hoffmann, 2003; Coles et al., 2012). Like its predecessors, TRPGne builds alternating fact and action layers that include all the possible applicable actions until a fixed point is reached, as described in Sections 2.9.1 and 2.9.2. This process is polynomial. Delete relaxation is also complete in the context of *reachability*. The TRPGne will reach the goal state for all states that can reach the goal state for the non-relaxed version of the problem, and any states that do not reach the goal state in the relaxed version of the problem will not reach the goal state in the non-relaxed version. The heuristic can be optimistic in some states where the goal state is reachable in the relaxed version and not in the real version.

Forward chaining search explores the state-space starting from the initial state, applying actions that are applicable and evaluating the result of their effects. This technique is sound, as the applicability and application of each action is verified and computed at each step, guaranteeing that a valid action sequence is maintained throughout the search process. Enforced Hill Climbing with Ascent Backtracking is not complete in itself, but a fall-back to conventional Weighted A* can make the search complete.

Linear programming is used to verify that a plan is valid with regards to the duration conditions of actions, ordering constraints and the trajectory of the values of time-dependent numeric fluents. This technique is known to be polynomial (Khachiyan, 1980; Karmarkar, 1984), and builds a feasible convex polytope from the variables and constraints, to find the optimal combination of values with respect to the objective function. A solution is only considered valid if the linear program for it produces a result, implying a valid schedule. This ensures that the solution is sound in terms of the temporal characteristics of the plan.

4.9 Planner Implementation

The algorithms described above were implemented in a new planner, referred to as DICE (Discrete Interference of Continuous Effects). This planner was implemented in Scala (version 2.11.7), a language that compiles to Java Bytecode and runs on the Java Virtual Machine.

4.9.1 Technology Platform Choice

Scala offers a good trade-off between performance and development productivity, to experiment with and prototype the proposed techniques. Naturally, programs written in a JVM language can be slower than those written in languages that compile to native machine code, such as C/C++ and Go. However, the productivity tools and mechanisms offered by the virtual machine, such as garbage collection and safe 64-bit memory access, offer significant advantages when prototyping new algorithms. Furthermore, Scala, a hybrid object-oriented and functional programming language, offers several in-built facilities on top of Java, that proved useful for our planner implementation. These include:

Lazy Evaluation: A mechanism that delays the evaluation of a function until it is needed, and caches the result for subsequent invocations. This is especially useful when evaluating

the heuristic value of a state, ensuring that it is not computed unless explicitly needed, and caching its return value for future use if it is required again from a different part of the program.

Parser - Combinator Framework: A concise API to build parsers within the source code of the program, with a syntax that is very similar to Extended Backus-Naur Form (EBNF). This framework provides a much more concise and maintainable alternative to other parsing frameworks, such as JavaCC or Flex/Bison. It does not require a pre-processing stage, since Scala supports custom Domain Specific Language definitions through operator overloading. This was used to implement the planner's PDDL parser.

Streams: These provide the mechanism to attach computation lazily to each element in an arbitrarily long sequence. This is especially useful when processing a large number of successor states, without having to pre-process them all or copy them in a separate collection. When performing EHC, a stream is used so that when a state that has a better heuristic value than the current best is encountered, the subsequent states are discarded without having to process them.

Functional programming: Higher order functions such as *map*, *filter* and *fold* provide the facilities to transform a collection of elements into another collection (such as mapping durative actions into its snap actions), filter out unwanted elements (such as invalid states), or combine elements together into one result (such as applying all the actions in a layer to a relaxed state, to obtain the relaxed state of the subsequent layer).

While a more native implementation could arguably be faster, the purpose of this work is to solve PSPACE and EXPSPACE-hard problems (Bylander, 1994; Rintanen, 2007). This fact makes the performance advantage from a native implementation less relevant. Furthermore, a more low-level implementation increases the risks of bugs and inefficiencies, especially to implement the mechanisms that are otherwise provided out of the box by high-level languages, such as lazy evaluation and streams. The Java platform also offers a very large ecosystem of libraries and APIs which can be used by the system. In this case, the Apache Commons Math library was used to compute linear programs, although this library can be swapped with other more production-grade LP solvers, such as CPlex, LPSolve or Gurobi.

4.9.2 Planner Architecture

The architecture of the planner is made up of four logical modules, the PDDL Parser, the Expression Pre-Processor, the Search module, and the Plan Builder.

The PDDL Parser processes the PDDL representation of the domain and problem file, creating an untyped and lifted abstract problem representation. The Expression Pre-processor then processes the domain types (including any type inheritance) and assigns the respective types to the actions and declared objects. Expressions are then normalised and actions are grounded, producing a grounded problem representation. This consists of the set of grounded actions, the initial state, the goal condition, any TILs and TIFs, and metric directives.

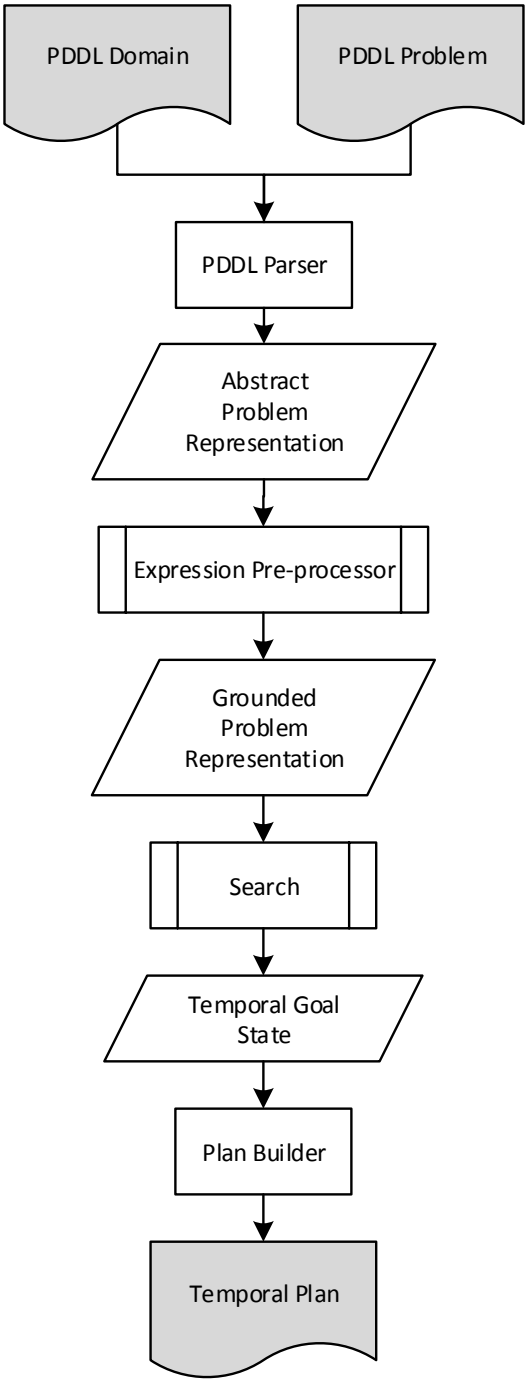


Figure 4.6: Planner Process Flow.

The Search module implements the search strategies described above to find a temporal state that satisfies the goal condition. For each temporal state, a linear program is computed to verify that it is feasible with regards to the temporal constraints, and to find a schedule for the partial plan to reach it from the initial state. Once a temporal state that satisfies the goal condition is found, it is passed to the Plan Builder module, which combines the happenings of the plan's steps with the schedule to output the solution in the required format. It also outputs useful statistics such as number of states explored and time taken to produce the solution. The full process flow of the planner is illustrated in Figure 4.6.

4.10 Evaluation

The proposed techniques are effective in domains that are rich in numeric and temporal characteristics. These include instantaneous and durative actions, variable duration expressions, and continuous numeric effects whose rate of change is not constant but subject to discrete updates. These updates can be caused either by discrete effects (start or end endpoints of durative actions and instantaneous actions) or TIFs.

For evaluation purposes, DICE was compared with a full hybrid PDDL-based planner, UPMurphi (Della Penna et al., 2009). While being the only PDDL-based planner known to support this class of problems, the time discretization approach used by UPMurphi suffers from loss of accuracy and scalability issues for plans with a long makespan.

UPMurphi (Della Penna et al., 2009, 2012) is a planner that evolved from a model checker. It is written in C++ and runs natively on Linux-based systems. Its approach to solve planning problems with continuous numeric effects is to discretize the time-line into time-steps. A domain and problem-specific planner is actually generated automatically from the PDDL definition, using a tool supplied with the UPMurphi package. The user can specify the value of the *quantum* (size of the time-step). UPMurphi uses breadth-first search to explore the state space and find the solution. It uses a 32-bit compilation, which unfortunately limits the maximum amount of memory to 3.2Gb, including the planner's internal overhead. The experiments below were performed using UPMurphi 3.1, the latest version at the time of writing.

The following is a description of the domains that were used, together with the results from executing both planners. Tests were performed on an Intel ® Core™ i7-3770 CPU @ 3.40GHz. The maximum amount of memory allocated to both planners was 3GB. Missing timings indicate that the planner ran out of memory or failed to yield a solution.

4.10.1 Project Planning Domain

This domain models a project that is composed of a number of tasks. To perform a task the appropriate resource that is skilled to do the task needs to be available and allocated to the resource. Each resource is only willing to work at certain times of the day. Furthermore each resource is paid at an hourly rate for the first 8 hours, but is ready to work an extra 2 hours at a premium overtime rate. Each resource can only perform one task at a time, and each resource can take a different amount of time to perform the same task. Each task can also have dependencies on other tasks. A task can depend on one or more tasks to be finished first before it can be started. The goal is to complete all the tasks, choosing the appropriate resources to perform the tasks, while accounting for the makespan or total cost as the metric objective. This domain has various interesting temporal and numeric aspects. The rate at which each resource is paid can change (due to overtime) while performing the task, and the time required to perform each task depends on the selected resource.

Listing 4.1 shows the actions of this domain. The `perform-task` durative action performs a task using a resource that `can-perform` it, given that it `can-work` throughout the execution of the action, the task does not have any other dependencies, and the resource is not already `occupied`. Its effect is that the task is marked `complete` at the end of the action, while increasing

the cost by the rate charged by that resource. The resource is marked as `occupied` throughout the execution of the action. The full PDDL domain definition is provided in Listing A.1.

```
;perform task if it has no dependencies on other tasks
(:durative-action perform-task
  :parameters (?r - resource ?t - task)
  :duration (= ?duration (time-required ?r ?t))
  :condition (and (at start (not (occupied ?r)))
                  (at start (not (complete ?t)))
                  (at start (no-dependency ?t))
                  (at start (can-perform ?r ?t))
                  (at start (can-work ?r))
                  (over all (can-work ?r)))
  :effect (and (at start (occupied ?r))
               (at end (not (occupied ?r)))
               (at end (complete ?t))
               (increase (total-cost) (* #t (cost ?r)))))

;perform a task if its dependency has been completed
(:durative-action perform-dependent-task
  :parameters (?r - resource ?pt ?t - task)
  :duration (= ?duration (time-required ?r ?t))
  :condition (and (at start (not (occupied ?r)))
                  (at start (not (complete ?t)))
                  (at start (dependency ?pt ?t))
                  (at start (complete ?pt))
                  (at start (can-work ?r))
                  (at start (can-perform ?r ?t))
                  (over all (can-work ?r)))
  :effect (and (at start (occupied ?r))
               (at end (not (occupied ?r)))
               (at end (complete ?t))
               (increase (total-cost) (* #t (cost ?r)))))

;completes a "dummy" milestone task if two tasks are complete
(:action join-tasks
  :parameters (?t1 ?t2 ?m - task)
  :precondition (and (milestone ?t1 ?t2 ?m)
                     (not (complete ?m)))
                     (complete ?t1)
                     (complete ?t2))
  :effect (and (complete ?m)))
```

Listing 4.1: Actions from the Project Planner domain.

The `perform-dependent-task` durative action works in a very similar way, but it checks that the parent task, `?pt`, which task `?t` depends on, is already marked `complete`. Finally, the `join-tasks` instantaneous action marks a `milestone` task `?m` as complete if two dependent tasks, `?t1` and `?t2`, are complete. A milestone task is a dummy task on which other tasks can depend. This allows tasks to have an arbitrary number of dependencies by chaining `milestone` tasks together.

Listing 4.2 shows a sample plan for the Project Planner problem instance in Listing A.2. In

this case both resources `r1` and `r2` only work from 9am till 7pm, so the first two tasks do not start before 9am. As soon as `r2` finishes `task4` can start, since `r2` is the only resource capable of performing the task. `task3` is dependent on two tasks, so a `milestone` task `m1` joins the two tasks so that `task3` can start. `task5` also depends on `task3` and `task4` being competed, but by the time `task4` finishes there is not enough time during that day to perform `task5`, so it is postponed to 9am of the next day (time 33).

```
9.0010: (perform-task r1 task1) [3.0000]
9.9950: (perform-task r2 task2) [3.0000]
12.9960: (perform-task r2 task4) [2.0000]
14.9970: (join-tasks task1 task2 m1)
14.9980: (perform-dependent-task r2 m1 task3) [4.0000]
18.9990: (join-tasks task3 task4 m2)
33.0010: (perform-dependent-task r1 m2 task5) [5.0000]
```

Listing 4.2: Sample plan for the Project Planner domain.

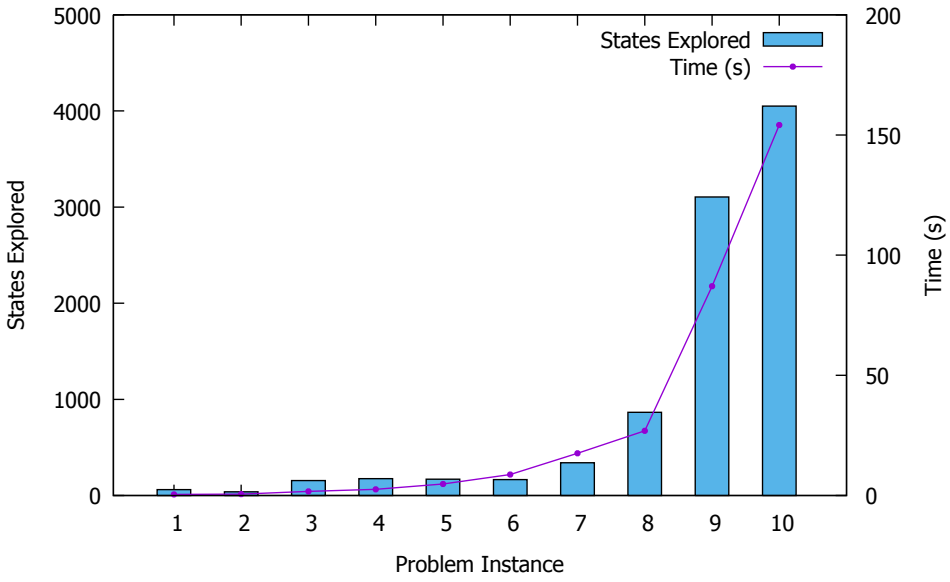


Figure 4.7: Time and Number of States Explored for Project Planner Domain.

Figure 4.7 and Table 4.1 show the results from solving 10 problem instances of the Project Planner domain. The *Tasks* column indicates how many tasks the problem involved, while the *R* column indicates how many resources were available to perform the tasks. This is only an indication of the problem size, because the complexity comes from the causal dependencies between tasks, and the association of resources with tasks (that is, which of the resources can actually perform the task). The *TH* column indicates the number of timed happenings in the problem, which corresponds to the time when the resources can start working, when they stop, and when their payment rate changes. Note that a problem spans multiple days. The *Time* column indicates how long the planner took to produce a result, while the *AB* column indicates whether *Ascent Backtracking* was activated (and thus EHC reached a dead-end). The *Plan Steps* column indicates the number of steps in the solution provided by the plan. (Each step corresponds to an instantaneous action, a start or end snap action, or a timed happening.) In this case, UPMurphi was set to use a quantum (size of time-step) value of 1 time unit.

Problem Instance					DICE			UPMurphi (1.0)	
#	Tasks	R	TH	Plan Steps	Time (s)	States	AB	Time (s)	States
1	2	2	4	8	0.45	62	Y	0.46	1,849
2	4	2	2	9	0.562	39	N	0.34	5,472
3	7	2	4	16	1.718	83	Y	31.48	879,740
4	8	3	5	19	2.55	128	Y	79.48	1,506,584
5	9	3	5	21	4.762	185	Y	•	•
6	10	4	4	22	8.751	228	Y	•	•
7	11	4	7	25	17.569	231	Y	•	•
8	12	5	5	27	26.897	865	Y	•	•
9	13	6	5	29	87.069	3105	Y	•	•
10	15	6	5	32	154.171	4050	Y	•	•

Table 4.1: Experimental Results for Project Planner Domain.

4.10.2 Intelligent Pump Control Domain

This domain models an intelligent pumping system in an industrial plant. Pumping of water for cooling or to operate turbines, together with pumping of other liquids (such as fuel or hydrolic fluid), is present in practically any industrial operation. An intelligent pumping system takes into consideration the characteristics of each pump within the circuit and also the required pressures to operate certain industrial processes. In our model, flow is directly dependent on these processes and the pumps present in the system. In reality, one must also account for restrictions that the fluid must pass through, such as friction, height in relation to the pump, and any tight bends that slow down the flow.

The interesting characteristic of this model is that with each additional industrial process that runs in parallel the flow in all the system for all the other industrial processes will go down. Similarly, if the pump in operation increases or decreases its flow, any running industrial processes at that point in time will experience a change in flow.

In this domain there are two types of industrial processes which require water at a certain pressure. A `fill-process` represents an industrial process that requires a container, such as a boiler, to be filled up to a certain level. A `use-process` represents an industrial process that requires a continuous supply of water for a certain duration, at a certain minimum flow. An example of this would be a cooling process, injecting water into the machinery of a plant and flushing it out to maintain temperatures within the required bounds. Each industrial process could also require that another process has been completed before it, represented by the `dependency` predicate, or that another process is running concurrently with it, represented by the `conc-dependency`.

```
(:types pump ind-process - object
        fill-process - ind-process
        use-process - ind-process)
(:constants plant - ind-process)
```

Listing 4.3: Types and Constants of the Intelligent Pump Control Domain

The domain model itself introduces a constant industrial process called a `plant`, which is set to both `running` and `complete`, so that processes that do not have any dependencies can

have the `plant` as their dependency (thus minimising the need for extra actions). This model also makes use of type inheritance, such that both `fill-process` and `use-process` can be used as an `ind-process`, representing a generic industrial process. This is shown in Listing 4.3.

```
(:durative-action fill
   :parameters (?dep ?conc-dep - ind-process ?f - fill-process)
   :duration (>= ?duration 0)
   :condition (and
     (at start (not(complete ?f)))
     (at start (not(running ?f)))
     (at start (dependency ?dep ?f))
     (at start (conc-dependency ?conc-dep ?f))
     (at start (complete ?dep))
     (at start (>= (current-flow-rate) (flow-rate ?f)))
     (over all (running ?conc-dep))
     (over all (<= (current-flow-rate)
                   (max-excess-flow-rate ?f)))
     (over all (<= (current-volume ?f) (max-fill-volume ?f)))
     (over all (>= (current-flow-rate) 0))
     (at end (>= (current-flow-rate) 0))
     (at end (>= (current-volume ?f) (min-fill-volume ?f))))
   :effect (and
     (at start (running ?f))
     (at start (decrease (current-flow-rate) (flow-rate ?f)))
     (at end (increase (current-flow-rate) (flow-rate ?f)))
     (at end (not (running ?f)))
     (at end (complete ?f))
     (increase (current-volume ?f)
               (* #t (+ (current-flow-rate)
                        (flow-rate ?f))))))

(:action start-pump
   :parameters (?p - pump)
   :precondition (and (= (current-pump-rate ?p) 0))
   :effect (and
     (assign (current-pump-rate ?p) (min-pump-rate ?p))
     (increase (current-flow-rate) (min-pump-rate ?p))))

(:action increase-pump-flow
   :parameters (?p - pump)
   :precondition (and
     (>= (current-pump-rate ?p) (min-pump-rate ?p))
     (<= (current-pump-rate ?p)
         (+ (max-pump-rate ?p) (pump-flow-step ?p))))
   :effect (and
     (increase (current-pump-rate ?p) (pump-flow-step ?p))
     (increase (current-flow-rate) (pump-flow-step ?p))))
```

Listing 4.4: Selected Actions from the Intelligent Pump Control Domain.

Each industrial process has a minimum `flow-rate` required to operate, and also a maximum bound on the excess flow `max-excess-flow-rate` which, if exceeded, would damage the equipment. The intelligent pump system must make use of the available pumps to turn them on

and off, and also increase or decrease their flow rates, so that the required industrial processes take place, within the required bounds.

Listing 4.4 shows some of the actions in this domain. The `fill` action increases the volume associated with the `fill-process` with its minimum `flow-rate` together with any excess flow in the circuit, represented by `current-flow-rate`. Any discrete actions that have an effect on the `current-flow-rate` will thus have an impact on this continuous effect. This process stops when the `current-volume` exceeds the associated `min-fill-volume`, representing the minimum amount of water needed. The `start-pump` action starts a `pump` present in the system, increasing the `current-flow-rate` with the minimum rate, denoted by `min-pump-rate`. This can be increased further if required by the `increase-pump-flow` action. The domain also has a `use` action, which is similar to `fill`, but its duration is predefined by the process. The `decrease-pump-flow` action reduces the flow of a pump as long as it does not go below the `min-pump-rate`, and the `stop-pump` action stops the pump if the flow rate is close enough to the `min-pump-rate`. The full domain definition together with a sample problem are provided in Listings A.3 and A.4.

Listing 4.5 shows a sample plan for this domain. In this case process `u1` needs to run concurrently with `f1`, so the pump's flow had to be increased prior to starting the second process. Towards the end of the plan, the flow rate of the pump, `p1`, had to be reduced gracefully in steps before stopping the pump altogether. This is an example of a causal chain constructed from implicit and explicit numeric preconditions, rather than just propositional facts, for which the DICE algorithm is especially designed.

```
0.0000: (start-pump p1)
0.0010: (fill plant plant f1) [255.2441]
0.0020: (fill plant plant f2) [95.2381]
95.2411: (increase-pump-flow p1)
95.2421: (use f2 plant u2) [60.0000]
155.2431: (increase-pump-flow p1)
155.2441: (use plant f1 u1) [100.0000]
255.2461: (fill u1 plant f3) [58.8235]
314.0706: (decrease-pump-flow p1)
314.0716: (decrease-pump-flow p1)
314.0726: (stop-pump p1)
```

Listing 4.5: Sample plan for the Intelligent Pump Control Domain.

Table 4.2 shows the results from solving some problem instances of the intelligent pump control domain using DICE, while Figure 4.8 illustrates the relationship between the number of states and time taken to find a solution. While this domain features a lot of interacting activity, with each action affecting the flows of other running actions, the heuristic works very well and guides the planner to a solution very effectively. Due to a bug in UPMurphi v3.1 (its type pre-processing code does not handle PDDL type inheritance properly), a variant of the domain with four versions of the `fill` and `use` actions had to be created (one for each combination of `fill-process` and `use-process` dependencies). UPMurphi's time-step size had to be set to 20.0 in order for it to yield any results even for the simpler problem instances. Missing entries indicate that the planner ran out of memory before providing a plan.

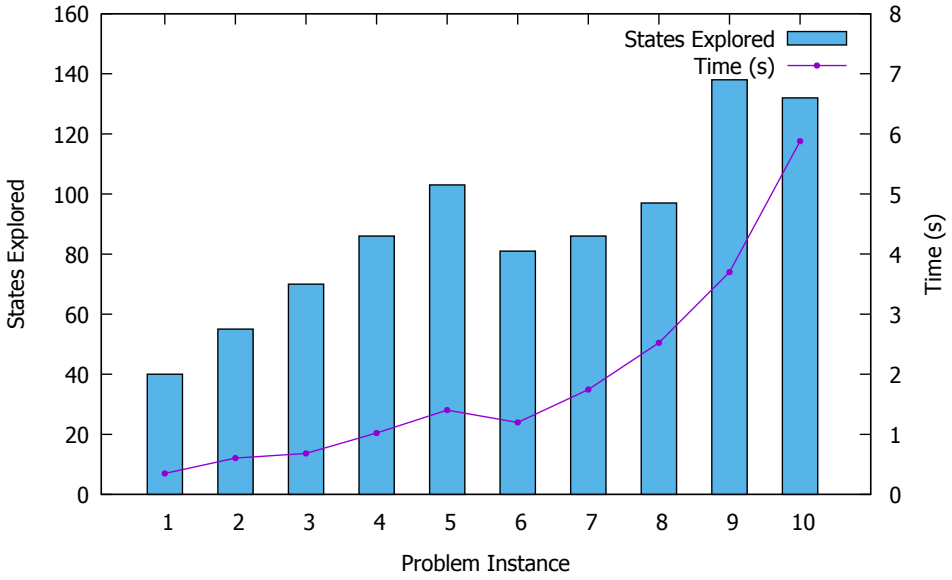


Figure 4.8: Time and Number of States Explored for the Intelligent Pump Control Domain.

Problem Instance				DICE			UPMurphi (20.0)	
#	Processes	Pumps	Plan Steps	Time (s)	States	AB	Time (s)	States
1	2	1	10	0.348	40	N	2.04	51,137
2	3	1	12	0.605	55	N	135.68	3,244,517
3	4	1	14	0.683	70	N	•	•
4	5	1	16	1.023	86	N	•	•
5	6	1	18	1.404	103	N	•	•
6	6	2	14	1.197	81	N	•	•
7	7	2	16	1.744	86	N	•	•
8	8	3	18	2.522	97	N	•	•
9	9	3	22	3.7	138	N	•	•
10	10	3	24	5.897	132	N	•	•

Table 4.2: Experimental Results for the Intelligent Pump Control Domain.

4.10.3 Planetary Rover Domain

This domain is a variant of the Rovers domain used in the International Planning Competition (Long and Fox, 2003a). The objective of this domain is to navigate to waypoints and perform scientific experiments at specific locations, collecting data from these experiments and transmitting them. The version presented below focuses on the power management aspect when operating the rover. The main source of power is the sun, which provides a different power level at different times during the plan. This is modelled through TIFs. The rover needs a specific minimum amount of power (in watts) to operate its main computer systems and sensors. Neither navigation nor experiments can take place if the rover is not fully operational. The rover only navigates between two waypoints when there is enough solar power throughout the whole journey. It is also possible to perform multiple experiments concurrently if they happen to be at the same location and enough power is available.

The rover is also equipped with a battery, which can be charged when there is solar power available, independently of whether the rover is operational. The data collected from experiments needs to be transmitted to an orbiter which relays it to Earth. The orbiter only passes over certain waypoints at certain specific time windows, and the rover needs to be at the corresponding waypoints during that time window to be able to transmit the data. This is modelled through TILs. Since these opportunities can coincide with times when there is minimal or no solar power, this operation relies solely on the rover's battery and does not require the rover to be operational.

This domain presents a complex suite of numeric characteristics and continuous effects, together with time windows, modelled through TIFs and TILs. An envelope action, `measure-power`, is clipped to the start of the plan, and is used to account for the changes in power levels throughout the plan. All other actions, such as `experiment`, require that the `measure-power` action is running, to keep track of the changes in solar energy. The `establish-uplink` is also an envelope action representing a communication session with the orbiter, during which `transmit-experiment-data` actions can be executed.

Listings 4.6 and 4.7 show these selected actions from this domain, while Listing 4.8 shows a sample plan. The `operate` action is another envelope action, which represents the rover being in a fully operational state, consuming power for its on-board computer and sensors, the `charge` action charges the battery, while the `navigate` action takes the rover from one waypoint to another. All these actions consume power in a similar way to the `experiment` and `transmit-experiment-data`. The full domain definition together with a sample problem instance are provided in Appendix A.

Figure 4.9 illustrates the performance of DICE on the Planetary Rover domain, showing the relationship between the number of explored states and time to solve each problem instance. Table 4.3 provides the details about each problem instance, with the W column indicating the number of waypoints, the O column indicating the number of objectives. Results from executing the same problem set on UPMurphi, with quantum value of 1.0 are also shown. Since the problem instances of this domain are designed to clip the `measure-power` action to the start of the plan by using a TIL, the time allowed to start the action had to be increased to match the quantum value. The same issue was also encountered for numeric fluents that

represented durations, such as `payload-transmission-duration` and `experiment-duration`. This exposes the limitations of the time discretization approach used in UPMurphi in domains that have rich numeric characteristics.

```
(:durative-action measure-power
  :parameters (?r - rover)
  :duration (>= ?duration 0)
  :condition (and (at start (not (measuring-power ?r)))
                  (at start (can-start-measuring-power ?r)))
  :effect (and
    (at start (measuring-power ?r))
    (increase (current-power ?r)
              (* #t (current-solar-power-dt ?r)))
    (at end (not (measuring-power ?r)))))

(:durative-action establish-uplink
   :parameters (?r - rover ?w - waypoint)
   :duration (>= ?duration 0)
   :condition (and
     (at start (rover-at ?r ?w))
     (at start (can-transmit-from ?w))
     (at start (not (charging ?r)))
     (at start (not (uplink-established ?r)))
     (at start (>= (battery-max-power ?r)
                   (power-needed-for-transmission ?r)))
     (at start (>= (battery-energy ?r)
                   (+ (establish-uplink-energy ?r)
                      (teardown-uplink-energy ?r))))
     (at start (measuring-power ?r))
     (over all (measuring-power ?r))
     (at end (measuring-power ?r))
     (over all (not (travelling ?r)))
     (over all (>= (battery-max-power ?r) 0))
     (over all (>= (battery-energy ?r) 0)))
   :effect (and
     (at start (uplink-established ?r))
     (at start (decrease (battery-energy ?r)
               (establish-uplink-energy ?r)))
     (decrease (battery-energy ?r)
               (* #t (power-needed-for-transmission ?r)))
     (at end (decrease (battery-energy ?r)
                       (teardown-uplink-energy ?r)))
     (at end (not (uplink-established ?r)))))
```

Listing 4.6: Selected Envelope Actions from the Planetary Rover Domain.

```
(:durative-action experiment
 :parameters (?r - rover ?w - waypoint ?o - objective)
 :duration (= ?duration (experiment-duration ?r ?o))
 :condition (and
   (at start (not (complete ?o)))
   (at start (not (running-experiment ?o)))
   (at start (objective-at ?o ?w))
```

```
    (at start (rover-at ?r ?w))
    (at start (operational ?r))
    (at start (>= (current-power ?r)
                  (power-needed-for-experiment ?r ?o)))
    (at start (measuring-power ?r))
    (over all (measuring-power ?r))
    (at end (measuring-power ?r))
    (over all (operational ?r))
    (over all (not (travelling ?r)))
    (over all (>= (current-power ?r) 0)))
  :effect (and
    (at start (running-experiment ?o))
    (at start (decrease (current-power ?r)
                        (power-needed-for-experiment ?r ?o)))
    (at end (increase (current-power ?r)
                      (power-needed-for-experiment ?r ?o)))
    (at end (not (running-experiment ?o)))
    (at end (complete ?o))))

(:durative-action transmit-experiment-data
  :parameters (?r - rover ?o - objective)
  :duration (= ?duration (payload-transmission-duration ?o))
  :condition (and
    (at start (not (transmitting ?r)))
    (at start (complete ?o))
    (at start (not (transmitted-data ?o)))
    (at start (measuring-power ?r))
    (over all (measuring-power ?r))
    (at end (measuring-power ?r))
    (at start (uplink-established ?r))
    (over all (uplink-established ?r)))
  :effect (and
    (at start (transmitting ?r))
    (at end (not (transmitting ?r)))
    (at end (transmitted-data ?o))))
```

Listing 4.7: Selected Durative Actions from the Planetary Rover Domain.

```
0.0000: (measure-power rover1) [18.3060]
6.2000: (operate rover1) [11.0000]
6.2400: (navigate rover1 w0 w1) [1.0000]
7.2410: (experiment rover1 w1 ob1) [2.0000]
9.2420: (navigate rover1 w1 w2) [2.0000]
9.2430: (charge rover1) [8.1570]
11.2430: (experiment rover1 w2 ob2) [2.0000]
11.2440: (experiment rover1 w2 ob3) [1.0000]
18.0010: (establish-uplink rover1 w2) [0.3040]
18.0020: (transmit-experiment-data rover1 ob1) [0.1000]
18.1030: (transmit-experiment-data rover1 ob3) [0.1000]
18.2040: (transmit-experiment-data rover1 ob2) [0.1000]
```

Listing 4.8: Sample Plan for the Planetary Rover Domain.

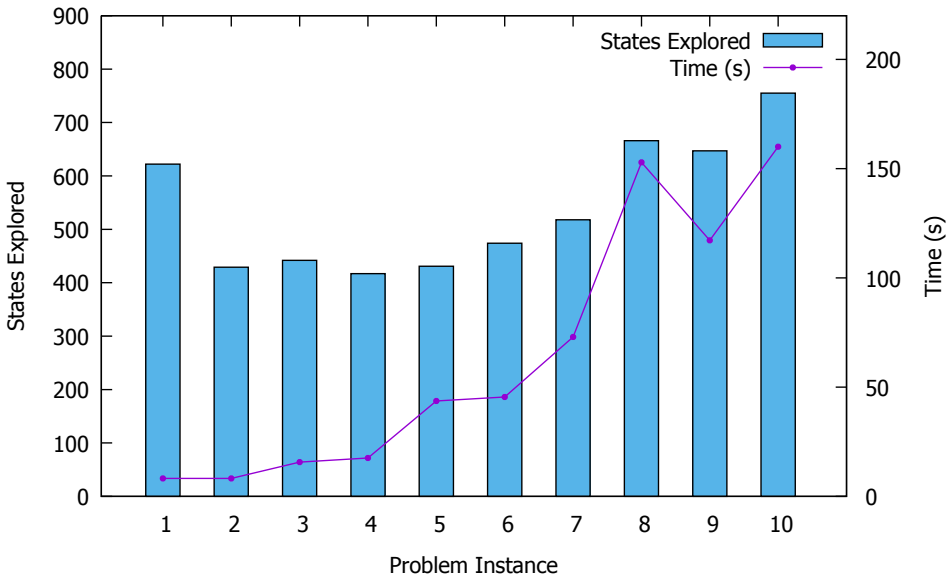


Figure 4.9: Time and Number of States Explored for the Planetary Rover Domain.

Problem Instance					DICE			UPMurphi (1.0)	
#	W	O	TH	Plan Steps	Time (s)	States	AB	Time	States
1	3	1	8	10	8.208	622	N	22.84	805,145
2	3	3	8	12	8.243	429	N	153.04	5,264,013
3	4	4	7	14	15.695	442	N	•	•
4	4	5	7	16	17.589	417	N	•	•
5	5	5	7	18	43.653	431	N	•	•
6	6	6	7	14	45.455	474	N	•	•
7	6	7	7	16	72.977	518	N	•	•
8	7	8	8	18	152.903	666	N	•	•
9	7	9	8	22	117.159	647	N	•	•
10	7	10	8	24	160.035	755	N	•	•

Table 4.3: Experimental Results for the Planetary Rover Domain.

4.11 Summary

Continuous effects are present in many real-world applications, and actions that interfere with their rates of change are also very common. This is of particular importance in problems where their solution features required concurrency. In this chapter an algorithm has been proposed to handle these kinds of planning problems. It builds on existing techniques that combine linear programming with forward-chaining search, but also supports constants in context. This can arise through either discrete effects of concurrent actions, or through TIFs. The main novelty of this technique is the capability to support a larger class of problems than is supported by current state-of-the art planners that make use of similar techniques. The delete relaxation heuristic based on the TRPG was also enhanced to take into account constants in context.

5. A PDDL Extension for the Semantic Attachment of External Modules

Real-world applications often involve complex numeric behaviour that takes into account several state variables and needs a rich expression language to model. Examples of these include power-flow equations, fluid dynamics, and trajectory computations. PDDL only supports simple arithmetic calculations, and one could argue that it operates at a higher symbolic reasoning level and that it should not be concerned with complex low level mathematical computations. However, the need to integrate high level planning reasoning with low level numeric computation is a common requirement in real-world problems.

A number of approaches can be found in literature to solve various aspects of this problem. These range from direct integration of the planner with an external solver (Piacentini et al., 2015) to extensions for platform-specific planner interfaces that perform *semantic attachment* for condition checking and discrete effect application (Dornhege et al., 2009; Hertle et al., 2012). On the other hand, Planning Modulo Theories (Gregory et al., 2012) defines a new planning definition language to introduce abstract data types, such as sets and lists, with externalised operations that can be accessed by the planning system.

In this chapter a different approach is proposed, which capitalises on the popularity and wide-spread use of PDDL. The advantage is that existing PDDL parsers will need very little modification to support this extension, making it easier to extend existent mainstream planners to incorporate this functionality. Any system written in any programming language can be enhanced to support this extension without making any changes to the module definitions or planning domains, and the problems that depend on them. The main purpose of this mechanism is to incorporate rich mathematical computations as part of the planning phase, also supporting continuous effects apart from discrete numeric effects and conditions.

5.1 A Review of Semantic Attachment Mechanisms

PDDL is the prevailing planning definition language. It is used in international planning competitions (McDermott et al., 1998; Fox and Long, 2003; Edelkamp and Hoffmann, 2004; Gerevini et al., 2009) for the deterministic planning category, and has evolved from supporting classical STRIPS (Fikes and Nilsson, 1971) formalisms, to more complex numerical and temporal constructs. However, being a high level reasoning language, its capabilities with regards to numerical expressiveness are limited when compared to general purpose

programming languages such as C++ and Java. These languages also benefit from external libraries that provide advanced mathematical functionality, together with I/O capabilities enabling input of sophisticated data structures to be used by a program. PDDL operates at a higher logical reasoning level that abstracts from this low-level complexity. However, the fact still remains that if automated planning technology is to be used in the real world, it needs to take into account complex numeric behaviour that governs the scenarios that are being modelled. This is in fact quite common in problems that involve physical phenomena such as power systems management (Piacentini et al., 2015), robotics (Dornhege et al., 2009), trajectory planning (Löhr et al., 2013) and fluid dynamics (Bogomolov et al., 2014; Shanahan, 1990).

One approach is to keep extending PDDL to support more sophisticated expressiveness and use universal hybrid planning techniques (Bryce et al., 2015; Della Penna et al., 2009) to solve these kinds of planning problems. However, without any support for subroutines and external libraries, this approach will always be limited to the level of mathematical expressiveness provided natively by the modelling language. Furthermore, domain independent hybrid approaches that support complex mathematical functions tend to be slow, since they depend on techniques such as time discretization, for which very few effective search guidance heuristics have been discovered to date.

An alternative approach is to keep the logical reasoning layer separate from the complex mathematical behaviour and create an interface between the two levels. This is known as *semantic attachment*, where symbols and objects have deeper semantics attached to them, than those apparent at the logical reasoning layer. Concretely, in PDDL, this implies that an object of a specific type may have additional information attached to it, which a specific solver would know how to interpret. Similarly, mathematical functions could also be embedded, which would be computed by the underlying external solver. The planner would interleave its planning process (traversing states in search of a plan) with calls to the external solver to compute the values of semantically attached state variables. Various flavours of this approach have been proposed, outlined below.

5.1.1 Domain-Specific Integration between Planner and Solver

This approach involves enhancing the planning system such that specific PDDL types or functions have a special meaning. The planning system invokes the external solver whenever it needs to interact with these special types. This interaction is planner and domain specific, customised to the scenario at hand. This approach has been used successfully to solve AC Voltage Control problems (Piacentini et al., 2015) using POPF-TIF, an enhanced version of the POPF temporal planner (Coles et al., 2010). The code of the planner was extended to incorporate the necessary power-flow equations and a special PDDL *encapsulated type*, in this case representing a power network, is used to represent the state of the network, which is hidden from the logical reasoning level modelled in PDDL. After each discrete state transition, the external solver is invoked to compute the respective underlying state of this *encapsulated type*.

The advantage of this approach is that it provides the full expressive power that can be expressed by the underlying general purpose programming language (in this case C++). The disadvantage however is that this approach is very planner specific. The user needs to have the

knowledge pertaining to the internal workings of the specific planning system to incorporate the necessary computation of the hidden state information underlying each encapsulated type.

5.1.2 PDDL/M - Dynamic Library Interface

This approach provides a more generic mechanism than the previous approach. It specifies a clear interface that modules need to implement, thus abstracting the module implementer from the internal details of the planning system. However, it is planner and language specific. PDDL/M (Dornhege et al., 2009) is an extension that allows the domain to directly specify the shared library implementation to include and which predicates or functions are provided by that module. Listing 5.1 shows a module's shared library being attached to a planning domain in PDDL/M.

```
(:modules
   (canLoad ?v - vehicle ?p - package
    conditionchecker canLoad@libTrans.so))
```

Listing 5.1: PDDL/M Module Defined in a Domain File. *Source: Dornhege et al. (2009)*

The advantage of this approach is that it allows multiple modules to be loaded dynamically. The disadvantage however is that this mechanism is planner and platform dependent. In fact it is very tightly coupled with the planner implementations for which it was designed, FF (Hoffmann and Nebel, 2001) and TFD (Eyerich et al., 2009).

The user or domain designer needs to be aware of the platform-specific details of both the planner and the module library being used. For example, the same domain cannot be used on the same planner compiled for Windows (which uses `.dll` files instead of `.so`) and the user will need to modify the PDDL file of the domain manually (which is supposed to be planner-independent), to work for the different platform. Similarly, if the planner was implemented in another programming language, such as Java™ or C#™, not only would the user need to modify the domain file, but also the proposed PDDL/M syntax for mapping predicates to functions would not work. This is due to the fact that global functions typically do not exist in these object oriented programming languages, and a way to specify the class which provides such method needs to be included.

Having such implementation-specific details permeating up to the abstract reasoning level is a general software engineering shortcoming which inherently limits the extent to which this approach can be adopted. Furthermore, the PDDL/M implementation in particular focuses only on condition checking, provided through *condition-checker* modules and discrete effects, provided through *effect-applicator* modules. Unlike the previous approach, there is no extra module-specific state information associated with the types specified in the domain or the objects in the problem instance. There is no way to specify which types the module interacts with or to validate the correctness of a domain in this regard. It is left up to the user to make sure that the parameters of the conditions or effects handled by the external module make sense. Finally, while this extension was designed for TFD, a temporal planner, the interface only handles discrete effects and provides no functionality for continuous effects.

5.1.3 Object-oriented Planning Language (OPL)

The Object-oriented Planning Language (OPL) (Hertle et al., 2012) was proposed as an improvement of the PDDL/M implementation, in order to address some of the shortcomings described above, namely the fact that using the PDDL/M framework is error-prone. OPL provides an object oriented way to define a domain which needs to access external modules. This is then pre-processed to generate the respective PDDL/M and module implementation stubs. OPL uses an object oriented C-style syntax, different from PDDL's Lisp-style syntax.

```
Domain RoomScanning {
  Type Pose {number x; number y; number th; }
  Type ScanTarget : Pose { boolean scanned; }
  Type Door {Pose approachPose; boolean open;}
  ... }
```

Listing 5.2: Example Domain definition in OPL. *Source: (Hertle et al., 2012)*

The advantage of this approach is that it provides a platform-independent syntax to specify domains, which are translated to the underlying implementation. The disadvantage of this is that a completely new planning language is being proposed, which could be a barrier to entry for most planning system implementations. In this case, OPL is translated to PDDL/M and the respective planner-specific module interface, which means that the generated PDDL/M domains are not portable across platforms. Another shortcoming of this approach is that the generated interface is domain-specific (Hertle et al., 2012). This means that reuse of modules is difficult and not part of the design philosophy of the approach. Each time a new domain is designed, the user needs to develop the planner-specific code to map the generated domain-specific interface to the necessary function invocations from the module library being used.

5.1.4 Planning Modulo Theories (PMT)

This approach was inspired by SAT Modulo Theories (Nieuwenhuis et al., 2006). In Planning Modulo Theories (PMT) (Gregory et al., 2012) classical planning is extended in a modular way, such that new types (possibly abstract data types such as sets and lists), can be added as modules, and functions acting on these types can be used by planning operators (actions). To a certain extent, these types are analogous to the encapsulated types used in the domain-specific integration approach. Two languages were created to support PMT (Gregory et al., 2012); the Module Definition Description Language (MDDL), used to define a module, and the Core Domain Description Language (CDDL), used to define a domain which makes use of PMT external modules. Both these languages use a Lisp-style syntax, somewhat similar to conventional PDDL. Listing 5.3 shows the definition of a `set` data type in MDDL, in terms of a generic type `a'`.

In each MDDL module, only one specific type can be defined. The focus of this framework is to extend PDDL with support for abstract data types which are defined in terms of generic (parametrised) types. This is somewhat analogous to *generic classes* in Java™and Scala, or *templates* in C++.

A planning domain that makes use of PMT modules is defined in CDDL. Listing 5.4 shows the header part of a domain definition in CDDL, that makes use of the `set` module defined

above. In this case functions such as `(at ?loc - location)` are not associated with a numeric value, as in PDDL 2.1, but with an abstract data type, `set`, representing a collection of unique `package` elements.

```
(define (module set)
  (:type set of a')
  (:functions
    (construct-set ?x+ - a') - set of a'
    (empty-set) - set of a'
    (cardinality ?s - set of a') - integer
    (member ?s - set of a' ?x - a') - boolean
    (subset ?x - set of a' ?y - set of a') - set of a'
    (union ?x - set of a' ?y - set of a') - set of a'
    (intersect ?x - set of a' ?y - set of a') - set of a'
    (difference ?x - set of a' ?y - set of a') - set of a'
    (add-element ?s - set of a' ?x - a') - set of a'
    (rem-element ?s - set of a' ?x - a') - set of a'
```

Listing 5.3: Definition of a set module in MDDL. *Source: Gregory et al. (2012)*

```
(define (domain setdomain)
  (:types
     location locatable - object
     truck obj - locatable)
  (:modules integer set)
  (:functions
     (at ?loc - location) - set of package
     (loc-of-truck ?tru - truck) - location
     (in ?tru - truck) - set of package
     (linked-to ?x - location) - set of location)
```

Listing 5.4: Header of a CDDL domain file. *Source: Gregory et al. (2012)*

The architecture of a PMT planner has a core and a set of modules, which provide the implementation of custom types defined in MDDL. This approach is arguably the most elegant of all the aforementioned techniques. Its focus is on supporting abstract data structures with parametrised data types, rather than supporting rich numeric computation, although one could imagine how this mechanism could be enhanced to work with rich numeric requirements and behaviours.

The main disadvantages with this approach have to do with the chosen syntactical structure of MDDL and CDDL. Each module in MDDL is restricted to define a single type, with operations acting on this type. In planning it is quite common to have multiple related object types that are cohesive in nature, and operations that have parameters involving multiple types. For example, a `grab` operation might require an `arm` type, representing the robotic arm, and a `grabbable` type, representing the object that needs to be grabbed. Furthermore, this approach provides no way to disambiguate between two modules which happen to have the same name.

5.1.5 Other Approaches

The need to attach richer semantic information to a system performing the high level reasoning is also common in robot task planning. Semantic information about the context within the world

that the system is operating in often involves sensory perception processing and also complex calculations related to computing localisation and geometric motion paths. Various approaches have been proposed for such scenarios, including integration of the planner into a Planning with Knowledge and Sensing framework (Gaschler et al., 2013; Petrick and Gaschler, 2014), and also combining a planner based on answer set programming with a low-level geometric reasoning and motion planning solver (Erdem et al., 2011; Aker et al., 2012). A similar approach is also used in other areas of A.I. such as Semantic Web, where declarative rules are integrated with external evaluations for semantic reasoning (Eiter et al., 2006).

5.1.6 PDDLx

We propose an approach that takes the best of each of the above approaches while addressing most of the shortcomings. It is somewhat similar in spirit to the mechanism proposed in PMT, but its main focus is on incorporating complex numeric computation within a planning system. Most of the PDDL syntax is retained, with encapsulated types representing abstract data types or data structures that carry more hidden information than what is visible at the PDDL reasoning level. Module predicates, functions and *methods* provide the necessary semantic attachment mechanism to perform computation by an external *class module*. This extension is referred to as PDDLx, which provides the necessary constructs to define a class module, specify its namespace for disambiguation, and declare the module's use within a planning domain. The result is an extension mechanism which:

(i) Extends PDDL in a platform-independent way, retaining the existing familiarity with the language and capitalizing on the effort invested in building robust PDDL parsers and pre-processors for the wide range of modern planners available written in several programming languages for various platforms.

(ii) Ensures portability of domains and problems across planners that provide a specific module, by specifying an implementation-agnostic definition of a module.

(iii) Supports multiple class modules, by providing a clear mechanism to achieve type-compatibility between objects of different modules, and restricting a module from accessing data that does not pertain to its defined types.

(iv) Promotes reuse of class modules by clearly separating the role of the domain designer, (the user of the module and planner, who does not need to have programming skills or knowledge of the intimate details of the planner) from that of the module developer (who provides an implementation of a specific module for a particular planning system).

(v) Supports a wide-range of functionality, including encapsulated types to which additional hidden information can be semantically attached, predicates and functions whose value is computed by the underlying class module implementation, and methods (equivalent to effect-applicators of PDDL/M).

(vi) Supports duration-dependent continuous functions that can be used for non-linear continuous effects of durative actions. This capability is not provided by any of the aforementioned approaches.

(vii) Provides a clear self-documented type-safe interface to the domain designer, which can be used by PDDL parsers, pre-processors and validators (Howey et al., 2004) to verify that a domain, which makes use of one or more class modules, is valid.

(viii) Allows module definitions to specify which data is read-only to the domain (computed only by the external class module), *initialisable* (assigned only through the initial state of the problem instance) or *mutable* (can be modified directly by an action's effect).

(ix) Encapsulates types, predicates, functions and methods in an object-oriented fashion (each module is a class providing these entities) while retaining the familiar PDDL syntax.

(x) Incorporates hints as to what predicates and functions are modified by a specific method, to help the planning system build the necessary data structures, such as relaxed planning graphs (Hoffmann and Nebel, 2001; Hoffmann, 2003) and causal graphs (Helmert, 2004), to search for a solution more efficiently.

PDDLx specifies how the interface to external modules should be defined (in a PDDL-like syntax), how domains can declare the use of such modules, and how problem instances can make use of them. PDDLx module definitions, domain definitions and problem instances are completely independent from the planner and platform it is running on. Two planners written in different programming languages for different operating systems will be able to process exactly the same PDDLx files without modification.

5.2 External Class Modules

As described above, PDDL was extended to support the concept of external class modules. We refer to the extended PDDL language supporting the use of external class modules as PDDLx.

A class module represents an entity, external to the planner, which provides a set of predefined object types, predicates, functions and methods, exposed as *members* of the *class*. These are defined in a class module definition which is the contract defining what the external class module is exposing to the PDDLx reasoning layer of the planner. The class module definition is used primarily in three phases:

Developing a new class module: It is considered good software engineering practice to define an abstract interface for two separate components that need to communicate with each other. A *contract-first* approach ensures that this separation of concerns is preserved and no implementation details leak through the interface. This approach also helps the class module developer to have full control over what is exposed to the PDDLx domain designer, in terms of which types, predicates, functions and methods.

Parsing and Verification: Some planning systems, such as TFD (Helmert, 2006), do not handle PDDL natively, and typically have a pre-processing phase that translates PDDL to some other representation. Having class module definitions available enables PDDLx parsers to process domains that make use of such modules, without actually having to interact with them. The class module definition also helps to verify the syntax of PDDLx domains and problems that make use of one or more class modules.

Multiple Implementations: It might be necessary for the same class module to be implemented for multiple planners, platforms or programming languages. Having a common class module definition ensures portability and platform-independence of domains that make use of such a class module. This approach also promotes reuse, potentially creating a library of external class modules that can be provided as add-ons to standard PDDL.

5.2.1 Class Module Development Process

A planning system that supports external class modules should be packaged with a tool (`pddlxgen`), which takes a module definition as input and generates the stubs for that class module with respect to the internal APIs of that planner. Depending on the mechanisms used by that planner to find solutions, additional planning life-cycle methods might be introduced. For instance, a relaxed version of a method might be required by planners that use relaxation heuristics in their search. Planners that support temporal planning with continuous effects might also pass a *time-elapsed* parameter to evaluate a continuous function at different time points. Once the stubs for the specific planner are generated from the class module definition, the class module developer can fill in the implementation details, potentially making use of other APIs providing the necessary black-box functionality. Figure 5.1 illustrates the development process of a new class module with the end result being a planner-specific implementation, packaged for the respective programming language and platform. This could for instance be a `.so` or `.dll` library, or a Java™ `.jar` file.

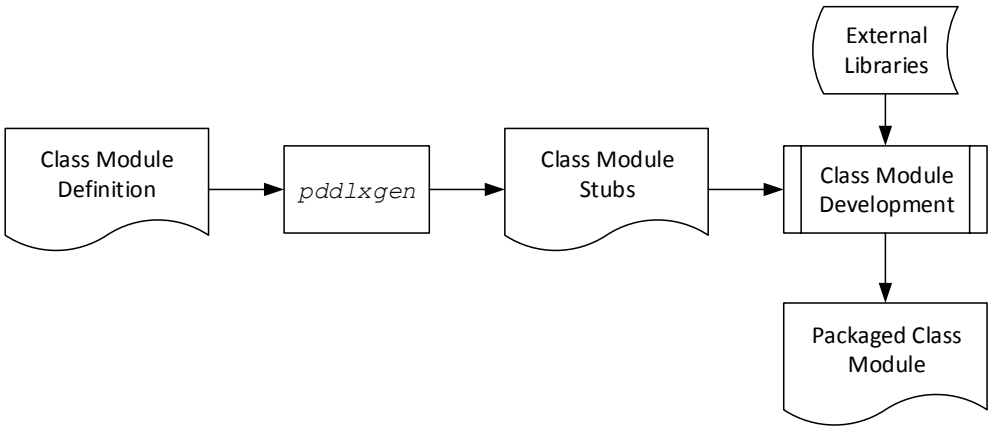


Figure 5.1: PDDLx Class Module Development Process.

To this end, we make a distinction between the respective roles of the *module developer* and the *domain designer*. The module developer is responsible for providing the library that encapsulates the respective functionality as a black box, implementing the necessary interfaces to integrate with the specific planning system. The module developer will naturally need to be skilled in developing software for the respective platform and programming language, and also have an insight into how the planning system works. On the other hand, the domain designer is the user of the class module, who will utilise the functionality provided to enrich the PDDL models and problem instances. While this might very well be the same person, especially for very problem-specific module implementations, the distinction is still imperative to allow distribution and reuse of common class modules by users who know how to design

PDDL models but are not skilled in programming or do not have a full understanding of how the planner they are using works. This is also one of the differentiating factors between this approach and prior cruder approaches, such as PDDL/M (Dornhege et al., 2009).

5.2.2 Class Module Definitions

The syntax of a class module definition looks very similar to a conventional PDDL domain definition, with some minor differences. Instead of the keyword **`domain`** the class module is defined with the keyword **`module`**, followed by the fully qualified name of the class module. A class module definition is composed of definitions of the following entities:

(i) **`requirements`** - the standard PDDL requirements, such as `typing` and `fluents`.

(ii) **`types`** - the types introduced by the class module.

(iii) **`constants`** - the constant objects that are always present for every problem whose domain imports the class module.

(iv) **`predicates`** - the propositional facts managed by the class module.

(v) **`functions`** - the numerical state variables known to and/or managed by the class module.

(vi) **`continuous-functions`** - any time-dependent continuous functions provided by the class module.

(vii) A list of **`method`** definitions, representing black-box effects implemented by the class module. These can be used only as effects of actions.

Class Module Namespaces

If class modules are to be distributed and potentially reusable, they are bound to conflict, especially if they use commonly used names for their types, predicates, and functions. A namespace convention is being proposed, analogous to namespaces and packages used in other programming languages. The following regular expression template defines the proposed hierarchical namespace convention, which enforces at least 2 levels; the organisation to which the module belongs, and the module name.

```
<Organisation>(.<Subentity>)*[.<Project>](.<Feature>)*.<Module>
```

The *fully qualified module name* is defined using the above template definition which includes its full namespace hierarchy up to and including the module name. This is specified following the **`module`** keyword that opens the module definition. In the example shown in Listing 5.5, the fully qualified module name is `Kcl.Planning.APS.Storage`, which indicates that the name of the module is `Storage` defined for the APS project, by the A.I. Planning group at King's College London (KCL).

Class Module Types

A class module is likely to introduce new PDDL types. These are the main entities through which the logical planning domain is connected to the underlying data structure instances of the

module. The domain or problem designer (the user of the external module) will be able to use this type within the PDDL domain and problem definitions. In the example shown in Listing 5.5, a new type `battery` is defined.

Composite types built from different types (potentially from different class modules) can be specified at the domain level using *multiple inheritance*. This enables each individual class module to attach extra information to the PDDL object with that composite type.

Constants

A class module may introduce constant object instantiations of the specific types it introduces to a domain. This is equivalent to the constant definitions in a conventional PDDL domain. This constant object can be used in the problem file as any other object of the same type, and is accessed by prefixing the constant's name with the class module's *alias* that is specified in the PDDLx domain, as described in Section 5.3.2.

Predicates and Functions

A module is also likely to introduce specific predicates and functions (numeric fluents). These resemble the properties exposed by the module, associated with instances of the module types.

By default, these predicates and functions are *logically immutable* or *read-only*. Actions cannot modify these predicates or numeric fluents directly, but they can query their values and use them in precondition expressions or the rvalue of effect expressions. These predicates and functions will only be updated through the respective methods provided by the module, or derived through the state information, somewhat analogous to PDDL 2.2 derived predicates (Edelkamp and Hoffmann, 2004) but transparent to the domain designer.

It is however quite common that properties need to be initialised and set to some value provided by the initial state of the planning problem. These kinds of predicates and functions can be prefixed with the keyword **`init`** to indicate that they can be set once in the initial state of the problem (in the respective **`:init`** block of the initial state of the problem).

It is also possible to allow PDDL actions to modify predicates and functions directly if required. These need to be prefixed with the keyword **`mutable`**, which indicates that an action can update the predicate (making it true or false), or the value of a numeric fluent, directly.

Continuous Functions

A module might have numeric fluents whose values change with respect to time, rather than just with respect to a discrete effect. These fluents can be declared in the **`:continuous-functions`** block, which indicates that the value of that numeric fluent might change with respect to time, and further probing of the value needs to be performed by the planning system.

Depending on the planner's implementation and its internal interface with the class module, this might involve discretization of the function through sampling of its values at various intervals, or some other numerical method for computing the value of the numeric fluent at a specific point in time.

Methods

Methods correspond to state updating procedures provided by the module. When a method is invoked, a new subsequent state is generated. Methods are defined within a **`:method`** block, which defines the parameters of that method, and the predicates or numeric fluents that it affects. Methods can only be used within effect expressions of actions. This capability makes PDDLx actions higher-order state transition functions, since they can take class module methods as part of their state updating procedure, which can also behave differently according to the current state in the plan.

The effects that a method has on a predicate can be that it either makes it true, indicated by specifying the predicate, or that it makes it false, indicated by the **`not`** keyword preceding the predicate. If the effect could be one of both, depending on the current state, the keyword **`change`** can be used to indicate this.

Similarly, the effects that a method has on a numeric fluent can be that it either has an **`increase`** or a **`decrease`** effect. If the effect could be one of both, depending on the current state in a plan, the keyword **`change`** can be used to indicate this.

Effect declarations in methods are essentially hints to indicate which state variables are updated by the respective method. The main purpose of effect declarations is to help the planning system gain more introspective information about a method, when performing its combinatorial search to find a solution. This is especially useful when working with heuristics such as Metric-FF (Hoffmann, 2003) or analysing which actions could potentially make a numeric condition true or false.

In this work, methods are restricted to be used only at the discrete endpoints of durative actions (**`at start`** or **`at end`**), or as effects of instantaneous actions. Continuously executing methods could also be incorporated, by allowing them to be specified as continuous effects (without any time specifier). However, this would imply that any numeric fluent could be modified behind the scenes in relation to time, and the planning system would need to probe the values of these fluents to ensure that their values did not violate any `overall` constraints, and to determine the time-points at which any numeric preconditions of other actions were satisfied.

Primitive Types

Apart from the types introduced by the module, two new *primitive types* are being introduced, shown in Table 5.1. This enables predicates, functions and methods provided by modules to perform context-dependent evaluations based on boolean or numeric arguments.

Type	Description
Number	A floating-point double precision number such as a numeric fluent or duration.
Boolean	A value of *true* or *false* such as that of a propositional predicate.

Table 5.1: Primitive PDDLx Argument Types.

5.2.3 Continuous Functions and Numeric Effects

PDDL 2.1 (Fox and Long, 2003) and 2.2 (Edelkamp and Hoffmann, 2004), together with PDDL+ (Fox and Long, 2002, 2006), support continuous numeric effects. These correspond to continuous changes to the value of a numeric fluent in relation to the duration of a durative action or process. When modelling complex processes these continuous effects might also need to interact with external modules to include complex calculations in the system.

The standard mechanism through which continuous effects are applied in durative actions and processes is that the effect of an action can **`increase`** or **`decrease`** the value of a fluent at some rate. With external class modules, this rate can also change with respect to an externally computed function that has a variable value with respect to time. This rate could also be a continuous function.

Whenever a numeric fluent is evaluated from a durative action or process, the planning system needs to pass contextual information, such as the time elapsed within the scope (durative action or process) in which the external module is being invoked. In addition to this, the planner must also be able to verify inequality constraints on the expression in order to enforce *overall* conditions throughout the execution of a durative action or a process.

These continuous functions are defined within the **`:continuous-functions`** block. This way, the planner's `pddlxgen` generates the corresponding functions with an additional duration parameter, enabling temporal information to flow between the planner and the external class module.

5.3 The PDDLx Syntax

PDDLx is an extension of PDDL to allow the definition of external class modules, and their use within the respective domains and problem instances. For a definition of the extensions to the PDDL grammar introduced by PDDLx refer to Appendix B.

5.3.1 Defining PDDLx Class Modules

Listing 5.5 shows an example definition for a class module that encapsulates the mechanism of an electricity storage device. The class module introduces a new type `battery`, which the domain designer can use as a type of the objects within the PDDLx domain and problem definition.

In this example the effect of charging and discharging a battery depends on its current state of charge, capacity and a Peukert constant which determines the efficiency of a battery, together with the power it needs to charge. The properties of the battery are expected to be initialised from the initial state of the planning problem instance, and are thus annotated with the **`init`** keyword. On the other hand, the `discharge-power` depends on which appliances are connected to the battery at that time, and can be modified by any actions in the plan. The **`mutable`** keyword indicates that this property is safe to use as the lvalue of an effect in the domain's action schema. The methods `start-charging`, `stop-charging`, `start-discharging` and `stop-discharging` perform some internal mathematical computation that changes the state of a `battery`.

```
(define (module Kcl.Planning.APS.Storage)
  (:requirements :typing :fluents)

  ;types introduced by this module
  (:types battery)

  ;predicates (these are immutable)
  (:predicates
    (init (available ?b - battery)))

  ;numeric fluents
  (:functions
    (init (capacity ?b - battery))
    (init (peukert ?b - battery))
    (init (state-of-charge ?b - battery))
    (init (charge-power ?b - battery))
    (mutable (discharge-power ?b - battery))
    (utilisation ?b - battery))

  ;numeric fluents that change with respect to time
  (:continuous-functions
    ;the time-dependent gradient of the discharge rate of ?b
    (discharge-rate ?b - battery))

  (:method start-charging
     :parameters (?b - battery)
     :effects ((not(available ?b))
               (increase (charge-power ?b))))

  (:method stop-charging
     :parameters (?b - battery ?dur - Number)
     :effects ((available ?b)
               (decrease (charge-power ?b))))

  (:method start-discharging
     :parameters (?b - battery)
     :effects ((not(available ?b))
               (increase (discharge-power ?b))))

  (:method stop-discharging
     :parameters (?b - battery ?dur - Number)
     :effects ((available ?b)
               (decrease (discharge-power ?b))))
)
```

Listing 5.5: Example of a PDDLx Class Module Definition

5.3.2 Defining PDDLx Domains

A PDDLx domain definition is very similar to a normal PDDL 2.2 domain. In order to indicate that support for class modules is needed, `:class-modules` is added to the **`:requirements`** section. A **`:classes`** section can then be included prior to the **`:predicates`** section, indicating

which class modules are needed by that domain. An alias name is given to each class module, which is then used throughout the domain definition, wherever a type, predicate, function or method provided by that class module is used. Members provided by a class module are prefixed with its alias followed by a dot, similar to the *dot notation* used by other languages. These members can then be used like normal PDDL elements defined within the domain.

The standard PDDL semantics do not enforce an explicit ordering of action effects, and they are always assumed to apply to the state prior to applying the action. This is not a limitation in standard PDDL because the same expression can be re-used in multiple effects. However, in PDDLx, a method can have a black-box effect on one or more numeric state variables, which might be needed for other effects of the same action. The **sequence** keyword alleviates this problem by allowing effects to be specified inside a **sequence** block, explicitly specifying that such effects should be applied in sequence, with each subsequent effect operating on any state variables of the state obtained from the resultant state of the preceding effect. Multiple methods or effect expressions can be chained together thus specifying an explicit ordering within an action's effect. A **sequence** block can have two or more effect expressions. When the planner is applying the effects of an action with a **sequence**, the effects in that sequence need to be applied in the specified order.

```
(define (domain storage-withclasses)
  (:requirements :typing :durative-actions :fluents
                 :duration-inequalities :class-modules)
  (:classes Storage - KCL.Planning.APS.Storage)
  (:functions (current-power))

  (:durative-action charge
     :parameters (?b - Storage.battery)
     :duration   (> ?duration 0)
     :condition  (and
       (at start (Storage.available ?b))
       (at start (< (Storage.state-of-charge ?b)
                    (Storage.capacity ?b)))
       (at end   (= (Storage.state-of-charge ?b)
                    (Storage.capacity ?b))))
     :effect     (and
       (at start
         (sequence (Storage.start-charging ?b)
           (increase (current-power)
                     (Storage.charge-power ?b))))
       (at end   (decrease (current-power)
                 (Storage.charge-power ?b)))
       (at end   (Storage.stop-charging ?b ?duration))))
)
```

Listing 5.6: Example of a PDDLx Domain Definition.

Listing 5.6 shows an example domain that makes use of the `KCL.Planning.APS.Storage` module by associating it to the `Storage` alias within the **:classes** block. The durative action `charge` makes use of the `Storage.available` predicate, together with the `Storage.state-of-charge` and `Storage.capacity` numeric fluents.

The methods `Storage.start-charging` and `Storage.stop-charging` are invoked as part of the action's effects, together with a query to the `Storage.charge-power` to determine how much power is consumed by the charging action after the respective method is executed.

5.3.3 Defining PDDLx Problem Instances

A PDDLx problem instance definition is also very similar to a PDDL 2.2 problem instance. The domain it belongs to is declared in exactly the same way, and optionally the `:class-modules` requirement can be included in the requirements definition. Any constants, predicates or functions imported from a class module can be accessed through the alias defined in the domain's **`:classes`** declaration. The PDDLx pre-processor can validate the problem instance against the definition of the class module to verify that the entities exist and are allowed to be initialised (they are annotated with the **`init`** or **`mutable`** keywords in the class module definition). Listing 5.7 shows an example problem instance for the domain defined in Listing 5.6, which uses the class module defined in Listing 5.5.

```
(define (problem storage-prob1)
    (:domain storage-withclasses)
    (:requirements :typing :durative-actions :fluents
                   :duration-inequalities :class-modules)
    (:objects battery1 - Storage.battery)
    (:init
       (= (current-power) 0)
       (Storage.available battery1)
       (= (Storage.state-of-charge battery1) 30)
       (= (Storage.capacity battery1) 100)
       (= (Storage.charge-power battery1) 2000)
       (= (Storage.peukert battery1) 1.2))

    (:goal (and (= (Storage.state-of-charge battery1)
                   (Storage.capacity battery1))))
)
```

Listing 5.7: Example of a PDDLx Problem Definition.

5.4 Architecture of a Planning System with Class Modules

A class module is a standalone component that, given the current state, is capable of deriving facts (boolean predicates or numeric fluents), or producing a new state through one of its methods. The class module itself is stateless (with the exception of any global configuration information), and no assumptions should be made about the state exploration mechanisms of the planning system. A planner could indeed explore states out of order, or need to back-track to previous states.

When a planner needs to have information computed by the external class module, it will pass a representation of the current state being evaluated. This state is as such immutable, and is just used to provide context to the class module. Essentially a function provided by a class module can be seen as a *closure* (Landin, 1964), where the state corresponds to the

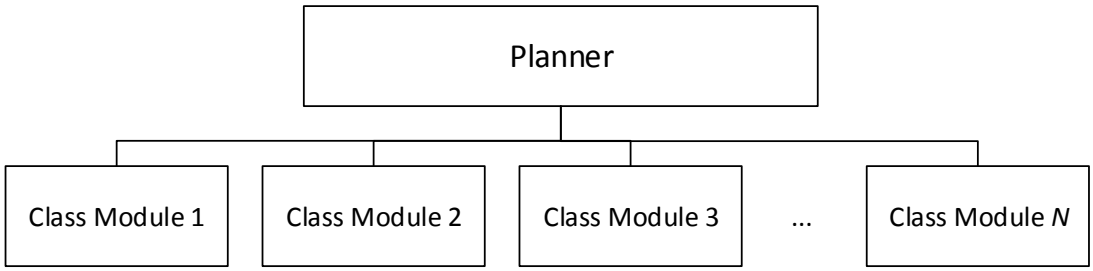


Figure 5.2: Architecture of a Planner with Class Modules

environment and the propositional and numeric state variables can be used as *free variables* by the semantically attached function or method.

5.4.1 Class Module Types

In order to guarantee data integrity and component isolation, a module should only have access to variables of its own types, and its own predicates and numeric fluents. This corresponds to those elements prefixed by the class alias of that module. The planning system should enforce this further by guaranteeing that the state representation a module receives is the relevant sub-set of the complete state representation. This way, each class module's variables are isolated from those of other class modules operating on the same domain. This is especially important when a domain makes use of multiple modules.

Formally, the types of each class module loaded by a PDDLx domain can be defined as follows. Let $\mathbb{M} = \{M_1, M_2, ..., M_n\}$ be the set of all class modules required by the PDDLx domain. Each module introduces its own implicit type, referred to as the *existential super-type* (Mitchell and Plotkin, 1988), which is the type from which all other types explicitly defined in the class module derive. Let T_i be the existential super-type of all types explicitly introduced by $M_i \in \mathbb{M}$. A type t is a sub-type of T_i, denoted $t <: T_i$, if it is either a type introduced by the module M_i, or a sub-type of one of such types. The <: operator refers to the transitive sub-type relationship between two types. When the planner queries information from a class module, M_i, for a state s, it should only pass a partial state, $s(M_i) \subseteq s$, that includes propositions and numeric state variables that are members of M_i. Naturally, the arguments of any predicates and functions of a class module, M_i, can only be of type $t_i <: T_i$, or one of the special primitive types `Boolean` or `Number`.

When multiple modules are required by a PDDLx domain, it is likely that the same object will need to be referenced by multiple modules. This case can be handled by creating a composite type by inheriting from the required module types, using multiple-inheritance, as shown in Listing 5.8. This enables objects of type `compositeType` to be passed as arguments to predicates, functions and methods expecting `typeM1` or `typeM2` as parameters.

```
(:classes Module1 - KCL.Planning.Module1
          Module2 - KCL.Planning.Module2)
(:types   compositeType - Module1.typeM1
          compositeType - Module2.typeM2)
```

Listing 5.8: Multiple Inheritance from Multiple Module Types.

5.4.2 Initialisation of Complex Data Structures

While the mechanism described in Section 5.2.2 may suffice for class modules that need only a handful of scalar parameters to be initialised in the initial state, some modules require more complex data structures such as matrices or large graph structures. A module might also have its own industry standard format to describe such domain-specific data structures. Examples of such data formats include CSV, Matlab and XML.

A class module can have its own specific way of loading such data from an external source, such as a file or database. It could also have its own configuration mechanism to specify which file to load or database to connect to. For example, a class module named `module1` could specify that a configuration file named `module1.cfg` needs to be provided, in which the data source and any additional module configuration information would reside. The planning system itself might also provide a mechanism to pass configuration information dynamically (through a global configuration file or command line parameters) to the individual class modules.

The class module will make any loaded data available through an encapsulated type. For example, a class module representing electricity storage might provide the type `battery-profile`. An object of this type, for example `high-capacity-profile`, would be provided by the class module and initialised from the specified data source. The name would depend on the convention adopted by the class module, but could typically either be a documented constant, or correspond to a unique identifier used in the class module configuration or in the proprietary data format. The PDDLx problem instance could then associate an object to the configuration type through a predicate provided by the class module, such as `(config ?b - battery ?bc - battery-conf)`. So, an object of type `battery` named `battery1`, could be associated with the battery configuration `high-capacity-conf` through `(config battery1 Storage.high-capacity-conf)` (where `Storage` is the alias of the class module set by the PDDLx domain).

5.5 Proof of Concept Implementation

A PDDLx reference implementation of the module stub generator, `pddlxgen`, was developed for the Java™ platform. Annotations supported by this specific programming language are useful to indicate the necessary mappings from the PDDL encapsulated types, predicates and fluents, to the respective Java methods. However, similar mechanisms can be developed for virtually any other general purpose programming language, including more native and highly performant ones such as C and C++. The planning system developed for this work was extended to support the loading of class modules to perform external computations and resolve module-specific state information. On start-up the system loads any PDDLx class modules and registers their fully qualified name. This is then used to perform class module lookup when a domain declares the class module in its **`:classes`** section.

In this implementation, the output of the class module stub generator, `pddlxgen`, is a Java interface, annotated with the respective PDDLx annotations. Listing 5.9 shows the interface code generated for the battery storage class module. The module developer would just need to implement this interface and include the compiled class in the Java classpath of the system.

```
@PddlxModule("Kcl.Planning.APS.Storage")
public interface Storage
  extends PddlxClassModule<Storage.StorageType>
{
  interface StorageType extends PddlObject {}

  @PddlxType(type="battery")
  interface Battery extends StorageType{}

  @PddlxPredicate(name="available", access = INIT)
  public boolean available(State state, Battery b);

  @PddlxFunction(name="peukert", access = INIT)
  public double peukert(State state, Battery b);

  @PddlxFunction(name="capacity", access = INIT)
  public double capacity(State state, Battery b);

  @PddlxFunction(name="charge-power", access = INIT)
  public double chargePower(State state, Battery b);

  @PddlxFunction(name="utilisation", access = READ_ONLY)
  public double utilisation(State state, Battery b);

  @PddlxFunction(name="state-of-charge", access = INIT)
  public double stateOfCharge(State state, Battery b);

  @PddlxFunction(name="discharge-power", access = MUTABLE)
  public double dischargePower(State state, Battery b);

  @PddlxContinuousFunction(name="discharge-rate",
                           access = READ_ONLY)
  public double dischargeRate(State state, double duration,
                              Battery b);

  @PddlxMethod(name="start-charging", effects = {
    @PddlxEffect(formula = "(available ?b)", effect = NOT),
    @PddlxEffect(formula = "(charge-power ?b)",
                 effect = INCREASE)})
  public State startCharging(State state, Battery b);

  @PddlxMethod(name="stop-charging", effects = {
    @PddlxEffect(formula = "(available ?b)", effect = AFFIRM),
    @PddlxEffect(formula = "(charge-power ?b)",
                 effect = DECREASE)})
  public State stopCharging(State state,
                            Battery b, double duration);

  @PddlxMethod(name="start-discharging", effects = {
    @PddlxEffect(formula = "(available ?b)", effect = NOT),
    @PddlxEffect(formula = "(discharge-power ?b)",
                 effect = INCREASE)})
```

```
  public State startDischarging(State state, Battery b);

  @PddlxMethod(name="stop-discharging", effects = {
    @PddlxEffect(formula = "(available ?b)", effect = AFFIRM),
    @PddlxEffect(formula = "(discharge-power ?b)",
                  effect = DECREASE)})
  public State stopDischarging(State state,
                               Battery b, double duration);
}
```

Listing 5.9: Java interface generated for a PDDLx Class Module.

At runtime, whenever a value needs to be evaluated from the external class module, the planner will generate the module-specific state representation, which is then passed to the module's respective class methods. In the case of predicates and functions, the methods that correspond to them return `boolean` or `double`, while in the case of class module methods their corresponding methods return a new `State` with the respective effects applied. The respective arguments correspond to the encapsulated types declared in the module definition, which will be passed by the planner. The additional annotations reflect the information in the PDDLx module definition. This self-documents the interface and also provide runtime hints to the planner about which predicates and numeric functions are affected by a method. The latter can be useful in problems with a large amount of grounded actions, and a fast method to find which are the applicable ones for a new state is required.

5.6 Summary

In this chapter, a new extension to the de facto standard PDDL has been proposed. This provides the mechanism for external class modules to be used in a planning domain. The main purpose of this mechanism is to support complex state-dependent numeric computations that would otherwise be difficult to model using just PDDL 2.1 constructs. The advantage of this approach is that the full expressive power of a general purpose programming language can be used.

A number of approaches have been used in the past to achieve the same purpose, such as integrating the planner directly with an external library (Piacentini et al., 2015), including the platform-specific library and function name directly in PDDL/M (Dornhege et al., 2009), a completely new programming language (OPL) which is translated into the respective platform-specific PDDL/M (Hertle et al., 2012), and the idea of encapsulating abstract data types as modules (PMT) (Gregory et al., 2012).

The proposed language, PDDLx, is designed to address the weaknesses of these prior approaches, using established software engineering principles, while retaining as much as possible of their advantages. The result is a framework that allows modules to be defined in a planner and platform independent way, such that they can be reused in multiple planning domains. The same module could have different implementations for a variety of platforms, programming languages and planners, while retaining the same module definition, allowing domains to be portable across planning systems. It is also possible that planners which adopt this framework start to support a standard set of modules, providing commonly needed functionality.

6. Planning with Non-Linear Continuous Monotonic Functions

Real-world temporal planning problems often involve non-linear continuous change to numeric variables. This is especially the case for scenarios that model the physical world. Examples of such instances include problems that involve fluid dynamics or power management. It is also difficult to model these problems in conventional PDDL since the constructs provided by the language to express continuous change (Fox and Long, 2003) are limited to arithmetic operators.

In this chapter we propose a mechanism through which a planner can take into account non-linear monotonic functions. This mechanism builds on the PDDLx extension introduced in Chapter 5, where the evaluation of certain functions is computed through an external evaluator called a class module.

An algorithm that approximates non-linear change by converging iteratively to a solution is proposed. Each semantically attached function can have a margin of error, within which the estimate of the non-linear continuous effect must lie. This non-linear iterative convergence extension, referred to as uNICOrn, was built into our planner implementation to support non-linear monotonic continuous behaviours.

6.1 Motivation

Non-linear continuous behaviour is common when the problem being modelled involves physical processes. One such example is Torricelli's Law, which approximates the outflow of a liquid by gravity from a small sharp hole in relation to the size of its container. This states that the velocity, v, with which liquid drains from a tank in relation to the height, h, of the surface of the liquid in the tank from the hole, is $v = \sqrt{2gh}$ (where g is gravity). Consequently, v will decrease as h decreases. Figure 6.1 illustrates this phenomenon for three holes at different heights.

While current temporal planners are quite effective in dealing with linear continuous change, most of them struggle when non-linear processes are introduced. It is in fact possible to model some non-linear functions in PDDL by having the rate of change of a numeric fluent being dependent on another continuously changing numeric fluent, but most planners are still not capable of handling such kind of models.

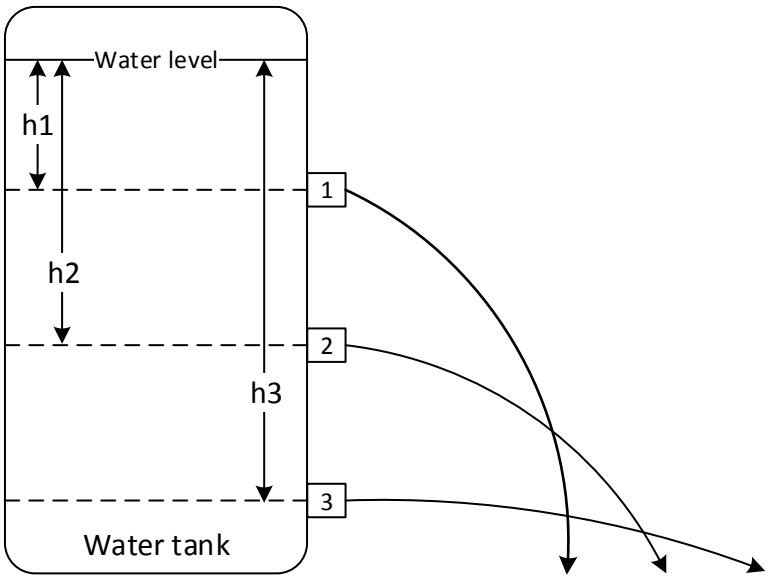


Figure 6.1: Torricelli's Law - The unassisted speed of efflux of a fluid from a sharp-edged hole depends on the height of the water level from the hole, which decreases over time.

6.2 Approaches to Planning with Non-Linear Behaviour

The need for planning with non-linear behaviour features numerous times in the literature. The behaviour of non-linear systems can be very complex and difficult to model. The latest prevailing approaches to solving non-linear planning problems are by either using time discretisation, or by modelling the non-linear behaviour using ordinary differential equations.

6.2.1 Time Discretisation

One of the few planners that does handle non-linear change is UPMurphi (Della Penna et al., 2009). As described earlier in Section 4.2.4, it uses an approach which involves time discretisation. A plan is found when the required conditions are satisfied for all discrete time points and the plan is then validated using a separate plan validator, VAL (Howey et al., 2004).

The advantage of this approach is that the continuous aspect of the non-linear function is abstracted away, and the problem is reduced to a number of points interpolating the function. However, the disadvantage with this approach is that it does not scale up very well, with long plans necessitating an increase in the size of each time-step to obtain a solution within a reasonable time. Larger time-steps not only introduce inaccuracy, but impose restrictions on the granularity of plan's schedule, since actions can only be applied at each discrete time-step.

6.2.2 Approximation using Ordinary Differential Equations

Another more recent approach (Bryce et al., 2015) involves combining DPLL-style SAT solving together with a differential equation solver, capable of integrating continuous effects formulated as Ordinary Differential Equations (ODE), and an Interval Constraint Propagation (ICP) solver. In this case, the user specifies the error precision within which the non-linear effects can be

evaluated. While this approach is promising, it is restricted to a set of well-behaved functions, using the native state-flow representation of the specific solver, *dReal* (Bryce et al., 2015). Even in this case, the main issue is scalability.

6.2.3 Linear Approximation

The alternative proposed in this work is to address the issue of scalability by establishing linear approximations between two time-points. This of course comes with some restrictions:

(i) The function is required to be monotonic within a temporal context (between two adjacent happenings).

(ii) Any constraints on the function need to be piecewise constant.

This technique is based on establishing a linear approximation of the non-linear function between two adjacent discrete time-points. The advantage of this approach is that it can leverage an efficient mathematical set of tools that handle linear functions, such as linear programming. It still treats each function as a continuous one, and does not require any time discretisation. Furthermore, it also handles actions of flexible durations, using a non-linear convergence algorithm to find the duration that satisfies the temporal and numeric constraints. This mechanism builds on the external class module framework proposed in Chapter 5, enabling continuous functions that satisfy the above constraints to be semantically attached to the model.

While the monotonicity assumption may seem restrictive, it turns out that a lot of real-world problems behave in this way. For example, filling a bucket from a tank never decreases the water level in the bucket, and charging a battery never decreases the state of charge of a battery. Furthermore, if the conditions that cause turning points of non-monotonic functions are known, discrete state changes could be introduced at those turning points, breaking up the non-monotonic function into smaller monotonic ones.

6.3 Planning with Non-Linear Continuous Monotonic Functions

A non-linear temporal planning task is modelled according to the PDDL 2.1 (Fox and Long, 2003) semantics, augmented with a set M of semantically attached functions provided by external class modules. As described above, it is assumed that any external continuous functions are monotonic within a discrete state. These kinds of functions are very common when modelling physical systems and were thus the primary focus of this work.

Let such a temporal planning task with semantic attachments be defined as $P_M = \langle \rho, \vartheta, M, O_{inst}, O_{dur}, s_0, G \rangle$, where:

- ρ is the set of atomic propositional facts.
- ϑ is the set of numeric fluents.
- M is the set of continuous monotonic functions available through semantically attached modules, taking the form $m : S \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}$, where S is the set of possible states. $m(s, t)$ maps the duration, t, spent in a state, $s \in S$, to the function's value at that time.

- O_{inst} is the set of grounded instantaneous actions.
- O_{dur} is the set of grounded durative actions.
- s_0 is the initial state of the problem, consisting of a subset of ρ and a mapping of ϑ to numeric values.
- G is a set of goal conditions that must be satisfied.

The continuous effects of a durative action can be linear changes of the form $v_{s'} = v_s + \mathrm{d}v \cdot t$, where v_s and $v_{s'}$ correspond to the value of variable v in states s and s' respectively, $\mathrm{d}v$ is the rate of change of v, and t corresponds to the duration spent in state s. Continuous effects can also be non-linear functions encapsulated through semantic attachment, of the form $v_{s'} = m(s,t)$, where $m \in M$. In the case of semantically attached non-linear effects, where the duration is not known, the rate of change $\mathrm{d}v$ will be determined through an iterative method that converges to a value within the required error bound.

6.3.1 Linear Approximation of Non-linear Monotonic Functions

Each function, $m(s,t)$, available through a semantically attached module, provides the value of that function, at a state s, for time $t \geq 0$. This is captured in the interface between the temporal planner and the external module. As explained in Section 5.4, a function available through semantically attached modules can be seen as a closure (Landin, 1964), where the state corresponds to the environment and the state variables can be used as free variables by the semantically attached function. The fact that these continuous functions are restricted to be monotonic helps to verify any piecewise constant invariant constraints that need to hold between two discrete states. Note that this is limited to cases where the sum of the effects on a variable, v, within a step i, Δv_i, is also monotonic, and the invariant condition is piecewise constant throughout its duration.

Theorem 1. For a continuous function $m : S \times \mathbb{R}_{\geq 0} \to \mathbb{R}$, conditioned by a state $s \in S$, there exists a linear function $\tilde{m} : \mathbb{R}_{\geq 0} \to \mathbb{R}$ for each $t \geq 0$, where $m(s,0) = \tilde{m}(0)$ and $m(s,t) = \tilde{m}(t)$.

Proof. Let $v_0 = m(s,0)$ and $v_t = m(s,t)$. There exists a linear function of the form $\tilde{m}(t) = \delta v \cdot t + v_0$ where $\delta v = (v_t - v_0)/t$, in which case $\tilde{m}(0) = v_0$ and $\tilde{m}(t) = v_t$. □

As described in Theorem 1, a non-linear continuous function can be intercepted by a linear one at the vertical axis (where $t = 0$) and at any other given time-point. For example, the monotonic function $y = t^3 + 2$ can be approximated linearly from $t = 0$ to $t = 2$ with the function $y = 4t + 2$.

If the desired duration of a continuous numeric effect is known beforehand, the linear function $\tilde{m}$ can effectively replace the non-linear one m and still provide the correct updated value for a continuously changing variable at time t. This reduction makes it possible to model and solve using an LP-based planning framework.

6.3.2 Non-Linear Iterative Convergence

In cases where the duration of a durative action is not known, the planner needs to find the right duration of an action such that the execution of the non-linear continuous function satisfies some constraint. For example, if the goal is to fill a bucket with water, the planner needs to find the right duration for a `fill` durative action, such that the required amount of water is supplied without overflowing or filling the bucket too little. In this case, the main challenge is how to determine the duration, and thus select the right linear approximation.

The approach proposed is to use an iterative method, that starts from computing δv_i, the gradient of v at step i, at one of the known bounds of the duration of the state, determined from its constraints C. It then iteratively improves its value until the error between the approximated change on v and the real one is less than or equal to a predefined error value for that function, e_m. The intuition behind Non-Linear Iterative Convergence is to start with an initial valid value of t and compute the non-linear function for that value, generate a linear approximation for it, and compute the linear program. If the difference between the obtained value and the actual one of the non-linear function is too large ($> e_m$), a new approximation is generated. Figure 6.2 illustrates this process. Starting from the value v_{ub} at one of the known bounds of the state's duration, ub, the algorithm interpolates a linear approximation (illustrated in orange) intercepting the non-linear function at $t = 0$ and $t = ub$. A linear program is used to determine the duration for the state considering the rest of the constraints, establishing it to be $d0$. At this point the error is computed from the value of the real function, and it is determined that it exceeds the maximum allowed error, e_m, for that function. A new gradient approximation is then used and the LP is computed once again, obtaining $d1$. The procedure is repeated until the error is small enough to accept the value (illustrated in green).

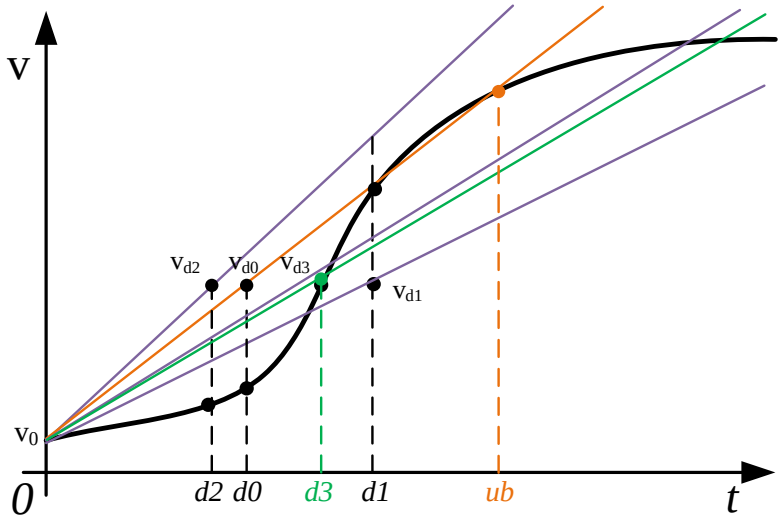


Figure 6.2: Non-linear Iterative Convergence on a Monotonic Continuous Function.

This process is described in detail in Algorithm 8. $\delta\tilde{V}$ maps the step index i, numeric fluent v, and continuous effect m to the approximated rate of change for that variable contributed from m. With each iteration, the difference between the approximated gradient and the actual

gradient is used to compute a new approximation, depending on whether the prior approximation overshoots the function or underestimates it.

Algorithm 8: Non-linear Iterative Convergence

Function `NonlinSchedule(`s, $\delta\tilde{V}$`)`

Data: A temporal state s and a set of initial approximations $\delta\tilde{V}$ of the rates of change for each non-linear effect m, on each variable v, at each step, i.

Result: A schedule for the plan to reach s, or Failure

converged $\leftarrow$ *true*

schedule $\leftarrow$ `scheduleWithLP(`$s, \delta\tilde{V}$`)`

if schedule $= \emptyset$ **then**

  return Failure

for $\langle i, v, m\rangle \in \Gamma_s$ **do**

  $d \leftarrow$ schedule$(i+1) -$ schedule(i)

  $\delta v \leftarrow m(s_i, d)$

  if $|(\delta v \cdot d) - (\delta\tilde{V}(i, v, m) \cdot d)| > e_m$ **then**

    converged $\leftarrow$ *false*

    if *((*$\delta v < 0$*) and (*$\delta\tilde{V}(i, v, m) < \delta v$*)) or ((*$\delta v > 0$*) and (*$\delta\tilde{V}(i, v, m) > \delta v$*))* **then**

      $\delta\tilde{v} \leftarrow (\delta\tilde{V}(i, v, m) + \delta v)/2$

      $s.C \leftarrow s.C \cup \{(d \leq t(i+1) - t(i))\}$

    else

      $\delta\tilde{v} \leftarrow \delta v - (\delta\tilde{V}(i, v, m) - \delta v)/2$

      $s.C \leftarrow s.C \cup \{(t(i+1) - t(i) \leq d)\}$

  $\delta\tilde{V}(i, v, m) \leftarrow \delta\tilde{v}$

if converged **then**

  return schedule

else

  return `NonlinSchedule(`$s, \delta\tilde{V}$`)`

Γ_s refers to the list of non-linear effects applied to the steps of the plan to reach the state s, with each element consisting of $\langle i, v, m\rangle$, where i is the index of the happening after which the non-linear continuous effect takes place (up to step $i+1$), $v \in \tilde{\vartheta}$ corresponds to the time-dependent numeric fluent updated by the effect, and m corresponds to an instance of a continuous effect involving a semantically attached external function.

The function `scheduleWithLP(`$s, \delta\tilde{V}$`)` sums up the rates of change on each variable $v \in \tilde{V}$ at each step, as defined in Equation 4.2. It combines these cumulative rates of change with the time-dependent preconditions of the actions in P and the temporal constraints C to build a linear program that computes a `schedule`, which is the sequence of timestamps for each step.

From this `schedule`, the actual durations of each state can be extracted, and the deviation of each approximated non-linear effect from its real value can be quantified. If this exceeds the maximum error defined for that effect, e_m, the flag `converged` is set to *false*, indicating that further iterations are needed in order to converge to the acceptable error threshold. A new gradient approximation for that continuous effect is then computed, and new temporal constraints on the duration of step i are added to $s.C$. These depend on whether the gradient

is negative or positive, and whether the approximation over-estimated or under-estimated the actual value. If all non-linear effects were estimated within the acceptable error threshold then the `schedule` is returned. Otherwise, the same procedure is called again recursively, with s carrying the new set of temporal constraints C and $\delta\tilde{V}$ updated with new approximations. The resultant plan from this procedure will be valid within the specified precision.

6.4 Adapting the Heuristic for Non-Linear Temporal Planning

Solving planning problems with non-linear continuous monotonic functions comes with additional computational complexity. With non-linear iterative convergence the linear program for a temporal state needs to be computed several times until values within the required accuracy are found. For this reason, having an efficient and effective heuristic is paramount in order to prune out states that are unlikely to lead to the goal.

6.4.1 Adding support for Non-Linear Monotonic Continuous Effects

The numeric-enhanced temporal relaxed planning graph (TRPGne) described in Section 4.6 can easily be extended to support non-linear continuous monotonic functions. Given that the rate of change between two states is assumed to be monotonic, the same applies between two layers in the TRPGne. The upper and lower bounds of the external non-linear function are computed, and depending on whether they increase or decrease (or both), the bounds of the affected numeric fluent are updated accordingly. The same procedure of analysing the TRPGne and extracting the relaxed plan takes place normally, as described in Section 4.6. However, this approach can be computationally very expensive. Performing *non-linear iterative convergence* on each numeric fluent to find its bounds can introduce significant computational overhead.

6.4.2 Backward Relaxation

Since the TRPGne considers both propositional and numeric state variables, and their trajectory over time as they are effected by discrete and continuous (potentially non-linear) effects, it proves to be very informative and can guide the search effectively to the goal. However, as the number of time-dependent variables in a problem increases, the computational cost of evaluating each state increases. One must keep in mind that for each time-dependent numeric variable, the upper and lower bounds have to be computed as the initial bounds for the first layer of the TRPGne. This can have a significant impact on the performance of the planning system, especially when these bounds need to be determined using non-linear iterative convergence.

On the other hand, it turns out that in a lot of planning problems that involve continuous numeric change, the durative actions responsible for such continuous effects often also include propositional and non-time-dependent state variables. These are typically used to indicate whether an activity has been performed. In models where these state variables are not present, it is normally quite easy to introduce a proposition for this purpose.

For this reason an alternative temporal relaxed planning graph that takes advantage of this fact is proposed. The user of the system can select which heuristic to use based on the

characteristics of the problem, although the possibility of detecting such properties automatically and selecting the appropriate heuristic function is not excluded.

The idea is to introduce *backward relaxation* into the heuristic evaluation of the current temporal state. That is, rather than computing the minimum and maximum bounds of time-dependent numeric variables, whose value is not yet known due to their dependence on the plan's schedule, their bounds are relaxed to $-\infty$ and ∞ respectively. The TRPGbr (*backward-relaxed* Temporal Relaxed Planning Graph) works exactly like the TRPGne but uses backward relaxation to exclude the need to compute the actual upper bound and lower bound of time-dependent numeric fluents. Non-time-dependent numeric fluents and propositional state variables are handled in the same way as in Metric-FF (Hoffmann, 2003).

This is of course less informative, but saves significant computational cost when the problem involves a high number of time-dependent numeric variables. This heuristic is most effective when the actions still carry non-time-dependent effects that lead to the goal.

In our implementation, we actually provide another variant, the TRPGnbr (*non-linear backward-relaxed* Temporal Relaxed Planning Graph), which only performs backward relaxation on non-linear numeric fluents. This eliminates the need to perform non-linear iterative convergence to find the bounds of these fluents, but still retains the actual lower and upper bounds on linear time-dependent numeric fluents, offering a good compromise for problems with time-dependent numeric goals.

6.5 Computational Characteristics

The techniques described in this chapter extend the algorithms described in Chapter 4, and carry forward most of the computational characteristics in terms of complexity, soundness and completeness, as discussed in Section 4.8. The TRPGbr and TRPGnbr have the same reachability characteristics of TRPGne, with the difference that they are more relaxed and less informative for time-dependent numeric fluents, thus being more optimistic.

The non-linear iterative convergence algorithm assumes that the input function is continuous and monotonic, thus restricting the set of possible functions to ones that converge using the proposed technique. Similar to other iterative approaches, such as the Newton-Raphson method or Gradient Descent, the speed of convergence depends on the shape of the curve (Hildebrand, 1987; Ortega and Rheinboldt, 1970).

6.6 Evaluation

The non-linear iterative convergence algorithm was implemented as an extension, called uNICOrn (Non-Linear Iterative COnvergence), developed on top of the planner described in Section 4.9. It uses the same planning framework, which includes support for external class modules, as described in Chapter 5. In order to evaluate the impact of non-linear iterative convergence, the planner can be set to run in either Breadth First Search (BrFS), EHC-ab with TRPGne, EHC-ab with TRPGbr (relaxing the bounds of time-dependent numeric fluents) and EHC-ab with TRPGnbr (relaxing only non-linear time-dependent numeric fluents).

The system was evaluated using the following domains, and compared with UPMurphi, the only non-linear PDDL-based planner. In the case of uNICOrn, the same non-linear characteristics were encapsulated in semantically attached functions. Tests were performed using an Intel® Core™ i7-3770 CPU @ 3.40GHz. Both planners were allocate a maximum of 3GB of memory, due to the 32-bit limitation of UPMurphi.

6.6.1 Tanks Domain

The Tanks domain consists of *tanks* and *buckets*, where each tank has a spigot at the bottom, through which the liquid inside the tank is allowed to flow out by gravity. The velocity with which the liquid flows out is not constant but follows Torricelli's law of fluid dynamics. This velocity decreases over time, in relation to the height of the liquid in the tank, until it reaches zero. The height itself also decreases non-linearly depending on the outflow velocity.

These two non-linear functions were semantically attached to the planning domain through `Torr.drain-rate` and `Torr.height-change`, where `Torr` is the alias of the module providing these external functions. The module also introduces a special type `Tank`, and numeric fluents for the `height`, `surface-area` and `hole-area`, representing the initial height of the fluid in a tank, the surface area of the fluid in a tank, and the area of the hole in the spigot, respectively. These are used by the module to calculate the drain rate and change in height according to Torricelli's law. A `Bucket` has a maximum `capacity` and `volume` of liquid it contains.

```
(:durative-action fill
 :parameters (?t - Torr.Tank
              ?b - Bucket)
 :duration (and (>= ?duration 0))
 :condition (and
   (over all (>= (Torr.height ?t) 0))
   (over all
       (<= (volume ?b) (capacity ?b)))
   (at start (not(filling ?b)))
   (at start (not(filled-from ?t))))
 :effect (and (at start (filling ?b))
    (at start (filled-from ?t))
    (increase (volume ?b)
      (* #t (Torr.drain-rate ?t)))
    (decrease (Torr.height ?t)
      (* #t (Torr.height-change ?t)))
    (at end (not(filling ?b)))))
```

Listing 6.1: The `fill` durative action.

Listing 6.1 shows the PDDL for the `fill` durative action, which updates the `volume` of the `Bucket ?b` together with the `height` of `Tank ?t`, in relation to the duration of the action. Throughout the action the `height` of the tank must remain non-negative, and the `volume` of liquid contained in the bucket must be less than or equal to its `capacity`. The goal of the problem is to fill the bucket up to within one hundred units of its capacity, and make use of all the tanks available. Listing 6.2 shows the goal condition with three tanks.

Listing 6.3 shows the output plan for the Tanks problem with 3 tanks, with error tolerance

$e_m = 0.001$ and $\varepsilon = 0.001$. Changing the error tolerance value will affect the durations of the actions in the plan since they have non-linear continuous effects.

```
(:goal (and
  (> (volume b1) (- (capacity b1) 100))
  (<= (volume b1) (capacity b1))
  (filled-from tank1)
  (filled-from tank2)
  (filled-from tank3)))
```

Listing 6.2: Goal condition for 3-tank problem.

```
0.0000: (fill tank1 bucket1) [10080.6340]
10080.6350: (fill tank2 bucket1) [10041.5222]
20122.1581: (fill tank3 bucket1) [10041.5353]
```

Listing 6.3: 3-tank plan with an Error Tolerance of 0.001

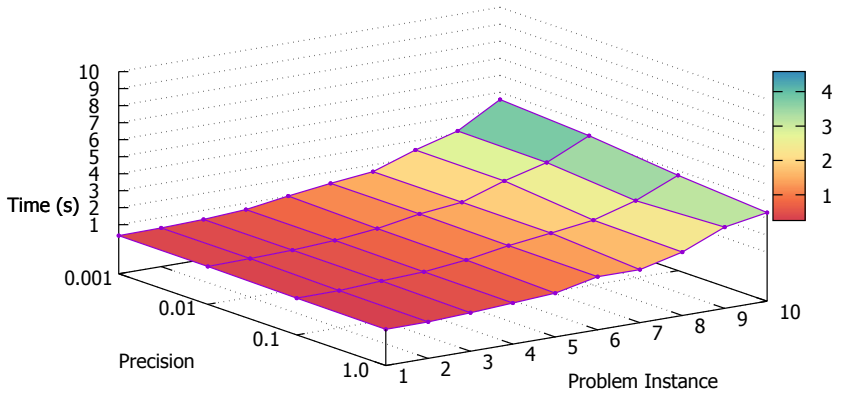


Figure 6.3: Performance on the Tanks Domain with EHC-ab using TRPGbr.

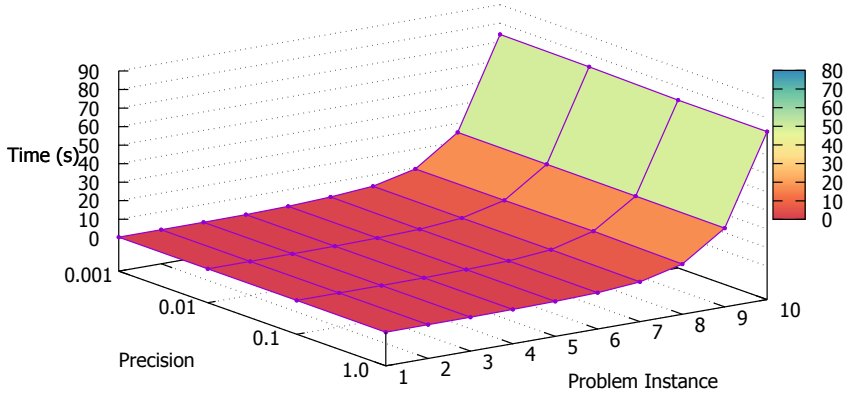


Figure 6.4: Performance on the Tanks Domain with Breadth First Search.

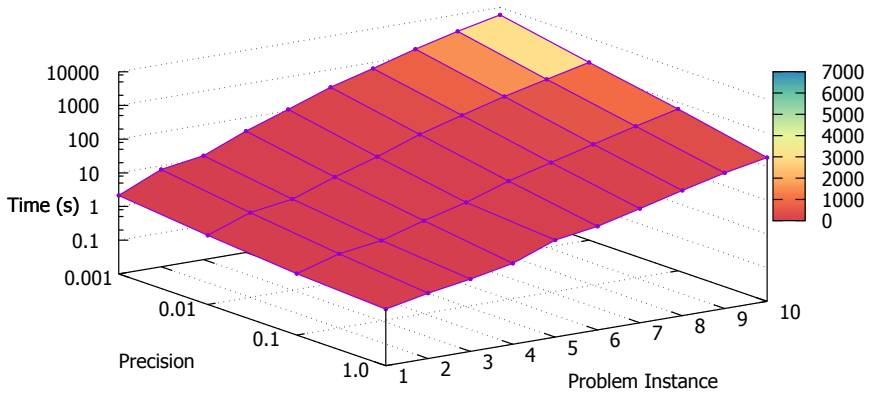


Figure 6.5: Performance on the Tanks Domain with EHC-ab using TRPGne.

Figures 6.3 to 6.5 show the time taken to produce a plan for this domain, for up to ten tanks, using EHC-ab with TRPGbr, BrFS and EHC-ab with TRPGne. (In this case, TRPGbr and TRPGnbr are equivalent since there are only non-linear time-dependent numeric fluents.) In each case the capacity of the bucket was set such that all tanks need to be used to achieve the goal volume. Each problem was also executed with a range of error tolerance values to analyse the impact of the increase in the number of iterations needed to converge.

As one can see from these results, EHC-ab with TRPGbr performed best, since it computes the heuristic evaluation function of each state very fast and still provides good search guidance. The reason for this was the propositional effect of the `fill` action. As one can see from Table 6.2, while BrFS explored significantly more states, the fact that it incurred no overhead to evaluate each state also outperformed EHC-ab with TRPGne, which although more informative, needed much more computation.

Tanks	EHC-ab / TRPGbr	BrFS	EHC-ab / TRPGne	UPMurphi
1	0.281	0.24	0.661	0.16
2	0.307	0.282	1.585	0.64
3	0.453	0.364	2.404	31.44
4	0.568	0.516	5.724	•
5	0.841	0.976	12.280	•
6	1.265	1.672	32.585	•
7	1.584	3.75	71.022	•
8	1.922	9.769	151.981	•
9	2.647	24.736	318.798	•
10	3.728	72.582	635.528	•

Table 6.1: Performance (in seconds) of uNICOrn (0.1 precision) and UPMurphi (with 1.0 time discretisation) on problem instances of the *Tanks* domain.

Tables 6.1 and 6.2 compare the performance of uNICOrn (using 0.1 error tolerance) with that of UPMurphi (using 1.0 time discretisation) using a maximum of 3GB memory on the same setup. Missing timings indicate that the planner ran out of memory. Listing 6.4 shows

Tanks	EHC-ab	BrFS	UPMurphi
1	7	7	52
2	13	19	10,473
3	21	51	1,479,909
4	31	131	•
5	43	323	•
6	57	771	•
7	73	1,795	•
8	91	4,099	•
9	111	9,219	•
10	133	20,483	•

Table 6.2: States explored using EHC-ab, BrFS and UPMurphi (with 1.0 time discretisation) on problem instances of the *Tanks* domain.

the `fill` action used in the version of the domain used by UPMurphi, which handles non-linear change in PDDL (since the square root function is not available in PDDL, the initial value is provided in the problem file through the `sqrtvolinit` fluent). Although UPMurphi also uses BrFS, its time discretisation approach forces it to explore many more states. On the other hand, the BrFS implementation in uNICOrn just explores the happenings where there are discrete state transitions, making it more efficient.

```
(:durative-action fill
  :parameters (?b - Bucket ?t - Tank)
  :duration (<= ?duration
                (/ (sqrtvolinit ?t) (flow-constant ?t)))
  :condition (and
    (over all (<= (bucket-volume ?b) (capacity ?b)))
    (at start (not (draining ?t)))
    (at start (not (filling ?b))))
  :effect (and (at start (assign (drain-time ?t) 0))
    (at start (assign (sqrtvol ?t) (sqrtvolinit ?t)))
    (at start (draining ?t)) (at start (filling ?b))
    (increase (drain-time ?t) (* #t 1))
    (decrease (tank-volume ?t)
      (* #t (* (* 2 (flow-constant ?t))
               (- (sqrtvolinit ?t)
                  (* (flow-constant ?t) (drain-time ?t))))))
    (decrease (sqrtvol ?t) (* #t (flow-constant ?t)))
    (increase (bucket-volume ?b)
      (* #t (* (* 2 (flow-constant ?t))
               (- (sqrtvolinit ?t)
                  (* (flow-constant ?t) (drain-time ?t))))))
    (at end (assign (sqrtvolinit ?t) (sqrtvol ?t)))
    (at end (not (draining ?t))) (at end (not (filling ?b)))))
```

Listing 6.4: The `fill` durative action of the Tanks domain used with UPMurphi.

6.6.2 Thermostat Domain

The Thermostat domain models a temperature control problem, where an object, typically a liquid in a container, needs to be kept within certain temperature bounds. This was inspired by the example shown earlier in Section 2.5. When direct heat is applied, the temperature increases linearly, but when heating is not being applied, the object cools down in a non-linear fashion. *Newton's law of cooling* approximates this as $\frac{dT}{dt} = -k(T_0 - T_a)$, where $T(t)$ is the temperature of the object at t, T_0 is the initial temperature of the object, T_a is the ambient temperature, and k is a cooling constant, which depends on the thermal characteristics of the object, such as specific heat capacity, mass of the object and any insulation properties.

The Thermostat domain represents a Bounded Trajectory Management Problem (BTMP) (Piacentini et al., 2015), where some numeric fluent of interest needs to be maintained within certain bounds for a specific planning horizon. This is different from the typical planning problems where the objective is to achieve some stable goal state.

This domain consists of two durative actions, shown in Listing 6.5. The `monitor` envelope action is clipped to execute at the beginning of the plan by the `can-start-monitoring` timed initial literal, and runs throughout the whole plan for the required duration specified by the `planning-horizon`. The `heat` durative action models the application of direct heat to the object of interest. Newton's law of cooling is encapsulated in an external class module, `Temperature`, and semantically attached to the uNICORn planner through the `Temperature.cooling-rate` fluent. This is computed from the current temperature, `Temperature.current-temp`, of the object, the ambient temperature `ambient-temp` (which could also change throughout the plan with timed initial fluents), and the `cooling-constant` associated with the object. The class module also introduces the type `Thermal` representing an object with the required thermal properties, as shown in Listing C.4.

```
(:durative-action monitor
    :parameters (?o - Temperature.Thermal)
    :duration (= ?duration (planning-horizon))
    :condition (and
      (at start (can-start-monitoring))
      (at start (not (monitoring ?o)))
      (over all (>= (Temperature.current-temp ?o)
                    (minimum-temp ?o)))
      (over all (<= (Temperature.current-temp ?o)
                    (maximum-temp ?o)))
      (at end (not (Temperature.heating ?o))))
    :effect (and
      (at start (monitoring ?o))
      (at end (not (monitoring ?o)))
      (at end (monitored ?o))
      (decrease (Temperature.current-temp ?o)
                (* #t (Temperature.cooling-rate ?o)))))

(:durative-action heat
   :parameters (?o - Temperature.Thermal)
   :duration (>= ?duration 0)
   :condition (and
```

```
      (at start (monitoring ?o))
      (over all (monitoring ?o))
      (at end (monitoring ?o))
      (at start (not (Temperature.heating ?o)))
      (at start (<= (Temperature.current-temp ?o)
                    (thermostat-on-temp ?o)))
      (over all (<= (Temperature.current-temp ?o)
                    (maximum-temp ?o)))
      (at end (>= (Temperature.current-temp ?o)
                  (thermostat-minoff-temp ?o))))
   :effect (and
      (at start (Temperature.heating ?o))
      (increase (Temperature.current-temp ?o)
                (* #t (heating-rate ?o)))
      (at end (not (Temperature.heating ?o)))
      (at end (not (Temperature.heating ?o)))))
```

Listing 6.5: The durative actions of the Thermostat domain.

```
(:goal (and (monitored o1)
   (>= (Temperature.current-temp o1) (minimum-temp o1))
   (<= (Temperature.current-temp o1) (maximum-temp o1))))
```

Listing 6.6: Goal condition for the Thermostat domain.

The `Thermal` object being monitored has an upper bound `maximum-temp` and a lower bound `minimum-temp` on the temperature. The `heat` action can be applied when the temperature of the object is less than or equal to the `thermostat-on-temp` and once started, cannot stop until at least the `thermostat-minoff-temp` temperature has been exceeded. The goal of the problem is to monitor the object's temperature, which will enforce the bounds throughout the plan, and also end with the temperature within the required bounds, as shown in Listing 6.6. Listing 6.7 shows a sample plan for this domain, where the required `planning-horizon` is set to 90 time units, and the planner opted to apply the `heat` action at specific periods to maintain the temperature within the required bounds. The full domain definition can be found in Listing C.5 together with a sample problem instance in Listing C.6.

```
0.0000: (monitor o1) [90.0000]
2.7177: (heat o1) [2.0000]
5.7189: (heat o1) [9.2821]
19.1363: (heat o1) [15.1856]
44.9990: (heat o1) [27.8885]
```

Listing 6.7: Sample plan for Thermostat Domain.

```
(:process cooling
  :parameters (?o - Thermal)
  :precondition (and (not (heating ?o)) (monitoring ?o))
  :effect (and
      (decrease (cooling-rate ?o) (* #t (cooling-constant ?o)))
      (decrease (current-temp ?o) (* #t (cooling-rate ?o)))))
```

Listing 6.8: Non-linear Cooling Process used in Thermostat Domain for UPMurphi.

```
(at start (assign (cooling-rate ?o) (* (cooling-constant ?o)
    (- (current-temp ?o) (ambient-temp)))))
```

Listing 6.9: Initialisation of the `cooling-rate` for the Thermostat Domain for UPMurphi.

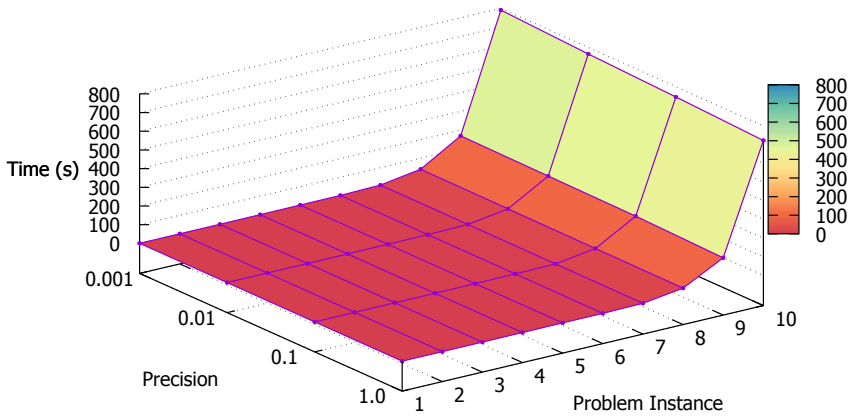


Figure 6.6: Performance on the Thermostat Domain with EHC-ab using TRPGne.

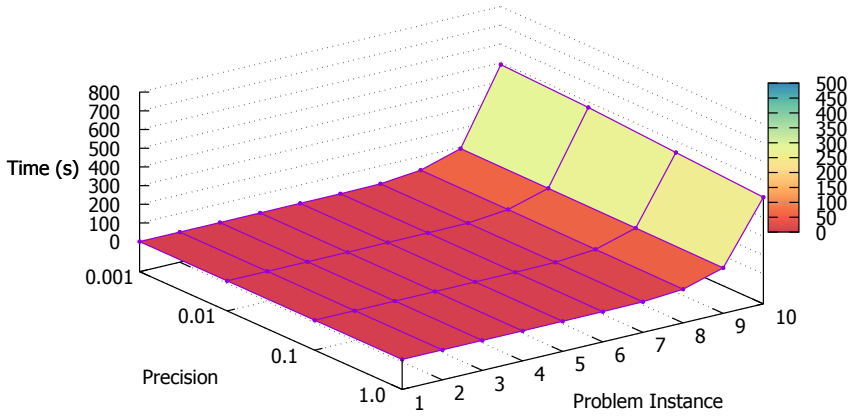


Figure 6.7: Performance on the Thermostat Domain with EHC-ab using TRPGbr.

Figures 6.6 to 6.9 show the time taken to produce a plan for problem instances of this domain, using EHC-ab with TRPGne, TRPGbr, and TRPGnbr respectively, together with BrFS. In each problem, more timed initial fluents were introduced to change the ambient temperature at specific timestamps of the plan, and the planning horizon was also increased, thus requiring more actions to maintain the temperature within the required bounds. As one can see, the TRPGbr and TRPGnbr heuristics were more performant, even though they were less informative when it comes to the non-linear continuous effects, since the amount of computation required to evaluate each state's heuristic value is much less.

Tables 6.3 and 6.4 compare the performance and number of states explored by uNICOrn (using 1.0 error tolorance) with those of UPMurphi (using 1.0 time discretisation), using a

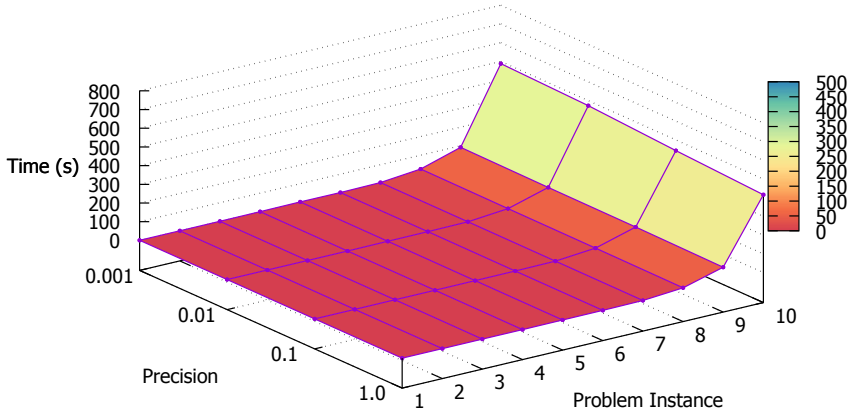


Figure 6.8: Performance on the Thermostat Domain with EHC-ab using TRPGnbr.

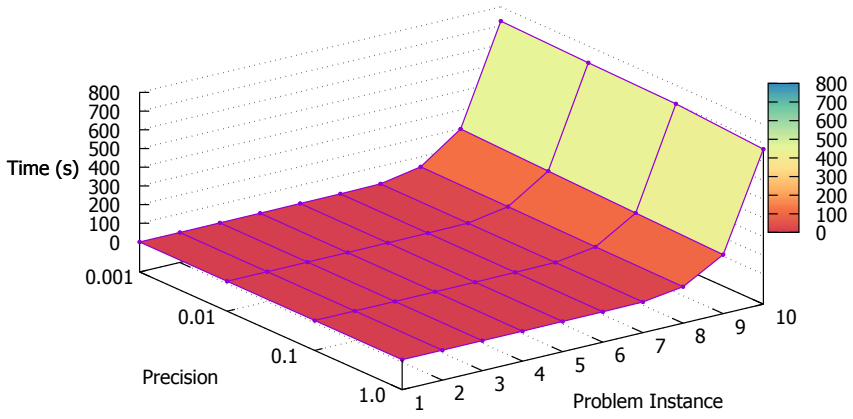


Figure 6.9: Performance on the Thermostat Domain with Breadth First Search.

maximum of 3GB memory on the same setup. In the case of UPMurphi, the non-linear cooling effect was modelled as a separate `cooling` process with two continuous effects, using PDDL+, as shown in Listing 6.8. The `monitor` action calculates the initial value of the `cooling-rate`, with the **`at start`** effect shown in Listing 6.9. The same effect is also applied **`at end`** of the `heat` action, so that the `cooling` process restarts with the correct `cooling-rate` value.

Problem Instance				uNICOrn				UPMurphi
#	M	TH	Plan Steps	TRPGne	TRPGbr	TRPGnbr	BrFS	Time (s)
1	30	1	7	0.327	0.304	0.321	0.162	0.54
2	40	1	7	0.363	0.29	0.29	0.158	5.16
3	50	2	8	0.389	0.325	0.394	0.33	39.72
4	60	3	9	0.572	0.468	0.47	0.477	523.24
5	70	4	12	1.556	1.135	1.154	1.235	•
6	80	5	13	1.659	1.279	1.274	2.181	•
7	90	5	15	4.667	3.205	3.257	4.222	•
8	100	6	20	34.868	20.044	20.481	34.651	•
9	110	7	23	146.241	83.096	79.639	155.096	•
10	120	8	26	722.814	410.316	416.419	668.477	•

Table 6.3: Performance (in seconds) of uNICOrn (1.0 precision) and UPMurphi (with 1.0 time discretisation) on problem instances of the *Thermostat* Domain.

#	uNICOrn EHC-ab	uNICOrn BrFS	UPMurphi
1	22	25	8,035
2	22	25	324,473
3	48	75	2,655,970
4	96	183	33,950,041
5	377	659	•
6	393	1,211	•
7	1,375	2,151	•
8	7,157	12,786	•
9	21,399	43,069	•
10	85,046	130,783	•

Table 6.4: States explored by uNICOrn (1.0 precision) and UPMurphi (with 1.0 time discretisation) on problem instances of the *Thermostat* Domain.

6.6.3 Non-Linear Planetary Rover Domain

The Non-Linear Planetary Rover is a variant of the linear Planetary Rover domain described in Section 4.10.3. The main difference is that battery charging follows a non-linear curve with diminishing returns. While the state of charge of a battery increases rapidly at the initial stages of charging, more time is needed to towards the end of the charging cycle to bring the battery to full charge. Figure 6.10 illustrates the charging curve of a typical battery.

This non-linear behaviour was encapsulated into an external class module providing electricity storage related functions, `Kcl.Planning.APS.Storage`, shown in Listing C.7. It approximates the charging curve using the Gauss error function $erf(t)$, scaled to

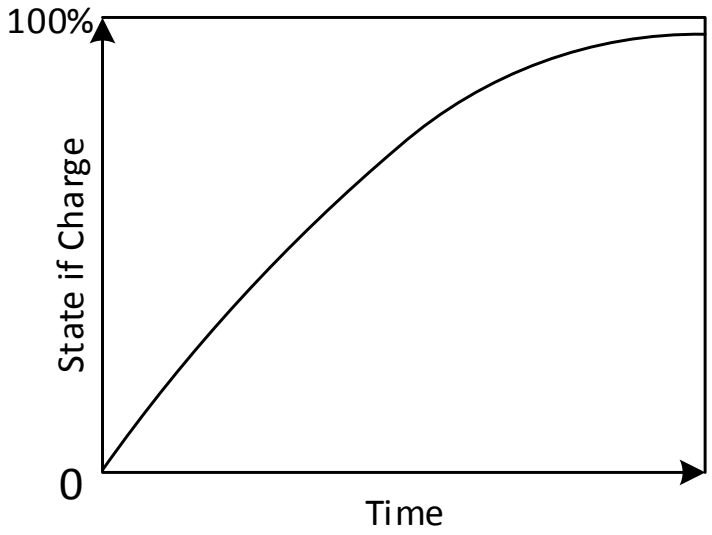


Figure 6.10: Typical Charging Curve for a Battery.

`full-charge-time`, the duration needed to fully charge the specific battery. Listing 6.10 shows the `charge` action of the Non-Linear Planetary Rover domain, where the `state-of-charge` of the `Battery` is increased with the non-linear `charge-rate` provided by the `Storage` class module (an alias of `Kcl.Planning.APS.Storage`). The rest of the actions and the goal of this domain are similar to the ones of the linear version. For a full domain definition and a sample problem instance refer to Listings C.8 and C.9. Listing 6.11 shows a sample plan for this domain.

```
(:durative-action charge
   :parameters
     (?r - rover ?b - Storage.Battery ?p - Storage.Profile)
   :duration (>= ?duration 0)
 :condition (and (at start (rover-battery ?r ?b))
  (at start (battery-charge-profile ?b ?p))
  (at start (>= (current-power ?r)
                (Storage.charge-power ?b ?p)))
  (at start (<= (Storage.state-of-charge ?b) 99))
  (at start (not (charging ?b)))
  (at start (measuring-power ?r))
  (over all (measuring-power ?r))
  (at end (measuring-power ?r))
  (over all (>= (current-power ?r) 0))
  (over all (<= (Storage.state-of-charge ?b) 100)))
 :effect (and (at start (charging ?b))
  (at start (increase (current-power ?r)
            (Storage.charge-power ?b ?p)))
  (increase (Storage.state-of-charge ?b)
            (* #t (Storage.charge-rate ?b ?p)))
  (at end (decrease (current-power ?r)
                    (Storage.charge-power ?b ?p)))
  (at end (not (charging ?b)))))
```

Listing 6.10: The `charge` action of the Non-Linear Planetary Rover Domain

```
0.0000: (measure-power rover1) [19.0090]
6.4000: (charge rover1 battery1 profile) [3.0040]
6.4010: (operate rover1) [11.3990]
6.4020: (navigate rover1 w0 w1) [1.0000]
7.4030: (experiment rover1 w1 ob1) [2.0000]
9.4050: (experiment rover1 w1 ob5) [1.0000]
10.4950: (navigate rover1 w1 w3) [1.0000]
11.4960: (experiment rover1 w3 ob4) [0.5000]
11.9970: (navigate rover1 w3 w2) [1.0000]
12.9980: (experiment rover1 w2 ob3) [1.0000]
12.9990: (experiment rover1 w2 ob2) [2.0000]
16.3990: (navigate rover1 w2 w5) [0.5000]
16.9000: (experiment rover1 w5 ob6) [0.3000]
17.2600: (navigate rover1 w5 w2) [0.5000]
18.0010: (establish-uplink rover1 battery1 w2) [1.0070]
18.0020: (transmit-experiment-data rover1 ob3) [0.1000]
18.1030: (transmit-experiment-data rover1 ob2) [0.1000]
18.2040: (transmit-experiment-data rover1 ob6) [0.2000]
18.4050: (transmit-experiment-data rover1 ob1) [0.1000]
18.5060: (transmit-experiment-data rover1 ob4) [0.1000]
18.6070: (transmit-experiment-data rover1 ob5) [0.4000]
```

Listing 6.11: Sample Plan for a Problem Instance of the Non-Linear Planetary Rover Domain.

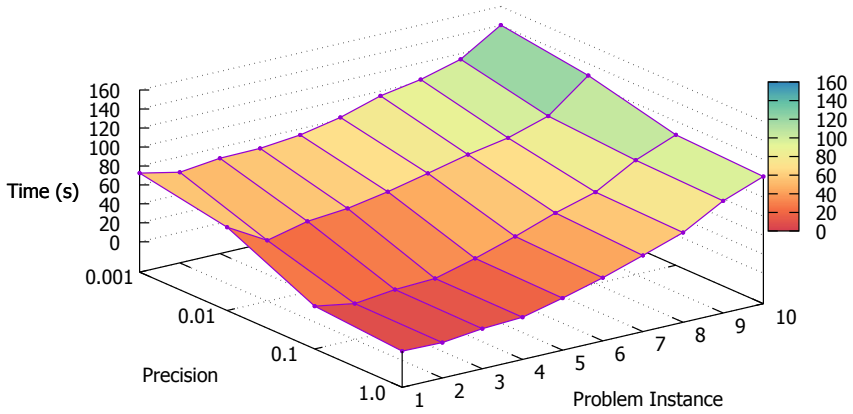


Figure 6.11: Performance of uNICOrn on Problem Instances of the Non-Linear Planetary Rover.

This domain is very rich in numeric constraints and dependencies, with actions only being applicable in the right power conditions. The changing levels of solar power complicate matters further, since applying a `navigate` or a `experiement` action too late in the plan could risk not having enough power (since solar power levels start to decrease after solar noon). For this reason, the full TRPGne had to be used to benefit from the increased informativeness of the heuristic. The faster backward relaxed versions led the planner to be lost in search, not yielding results in a timely manner for most of the problem instances. For this reason, only the results from the TRPGne were included, shown in Figure 6.11. Table 6.5 shows the detailed experimental results for this domain, with column W indicating the number of

waypoints, column *O* indicating the number of objectives, column *TH* indicating the number of timed happenings, and `Steps` indicating the total number of steps in the plan (including all snap actions and timed happenings). Since this domain uses the Gauss error function, $erf(t)$, to model the battery charging curve it was not possible to create a matching PDDL-only model for UPMurphi. This demonstrates the expressive power of PDDLx, enabling the domain designer to make use of much more complicated mathematical functions in the planning model.

Problem Instance					Performance of uNICOrn with Increasing Error Precision				
#	W	O	TH	Steps	States	1.0	0.1	0.01	0.001
1	3	1	8	24	177	6.181	13.087	55.833	72.613
2	3	3	7	31	226	5.386	6.002	32.125	63.633
3	4	4	7	37	326	10.262	10.913	42.441	68.569
4	4	5	7	41	339	12.48	12.895	46.172	69.113
5	5	5	7	45	414	22.877	24.204	54.185	73.529
6	6	6	7	49	424	34.499	37.606	63.721	82.221
7	7	7	7	51	448	48.234	52.515	73.719	95.167
8	8	8	8	60	526	62.551	64.822	81.515	102.551
9	9	9	8	66	594	86.083	88.646	94.698	113.998
10	10	10	8	68	668	102.155	105.513	127.552	140.255

Table 6.5: Experimental Results on the Non-Linear Planetary Rover using EHC-ab / TRPGne.

6.7 Summary

Non-linear change is present in many real-world contexts. We have presented an approach that builds on existing LP-based techniques, which are already quite effective at handling linear change, and enhances them with the capabilities to approximate monotonic non-linear change through an iterative improvement method. The planning framework is designed to support the inclusion of virtually any non-linear monotonic continuous function in a planning domain, through the semantic attachment mechanism proposed in Chapter 5. Furthermore, the proposed framework allows for the inclusion of continuous functions that cannot be easily expressed with conventional PDDL.

The TRPGne presented in Chapter 4 was also enhanced to include these continuous functions in the heuristic. Due to the increased computational complexity introduced by this mechanism, two more relaxed versions, TRPGbr and TRPGnbr, were also proposed to avoid the evaluation of the lower and upper bound of each time-dependent numeric fluent when evaluating the heuristic value of each state. While these can potentially be significantly less informative, as shown in the experiments performed on the Non-Linear Planetary Rover domain, they could be very effective in domains where the non-time-dependent effects already provide enough guidance information, as shown in the Tanks and Thermostat domains.

7. Case Study: Automated Demand Dispatch for Load Management

The main motivation for this work was to analyse the characteristics of the Automated Demand Dispatch problem for electricity load management, and research the planning techniques that could be applied to find solutions for this domain. In this chapter the details of this problem will be described, together with its mathematical optimisation model. The challenges presented by this model are highlighted, accompanied with proposals on how the techniques proposed in this thesis can help solve problem instances in this domain. This chapter concludes with further ideas on how the proposed approach can fit in a larger architecture of interconnected systems to work in a realistic setup.

7.1 Power Grids of the Future

The energy landscape is bound to change substantially in the next few decades. Power suppliers are expected to make use of clean and renewable energy sources. At the same time, they are expected to provide a secure and reliable energy supply to cater for the ever-growing dependency of the economy on electricity. Demand is also expected to increase, especially due to the electrification of services such as heating and transport. With the proliferation of home-based generation, such as photovoltaic panels, guaranteeing grid stability through the balance of supply and demand becomes more critical. These changes are likely to bring new challenges for the current electricity infrastructure.

Smart Grid technologies are considered to be crucial for a more intelligent network-wide power management strategy that can help address future capacity and regulatory requirements in a cost-effective manner (Ipakchi and Albuyeh, 2009). They support digital communication between all entities in the network, together with real-time metering and monitoring. This opens up various possibilities for better management and control of the electricity grid, and also enables the implementation of demand-response policies.

DSM (Strbac, 2008) refers to mechanisms through which the demand profile can be influenced. Some of the reasons a utility would want to shape the load curve could be to reduce peaks, increase the utilisation of generation resources during low demand periods, reduce the overall consumption, or encourage the use of electricity during times when higher volumes of cheaper (or cleaner) energy are available (Luo et al., 2010).

These mechanisms could be indirect or direct. The former uses tools such as time-of-

use tariffs, consumption-based tariff bands, discounts and customer education. This approach has the disadvantage of introducing more uncertainty since the consumer response is not guaranteed (Zehir and Bagriyanik, 2012). On the other hand, direct load control assumes that smart appliances and smart homes support remote actuation capabilities (through the smart grid communication infrastructure). In such a scenario, the operator can take over the operation of the respective device. The key issue with this latter approach is that it must be executed without having a significant impact on the user's lifestyle. This restricts the devices that can be used to those categorised as background loads (such as space and water heating, air conditioning, refrigeration and dehumidification) and schedulable loads (such as dish-washing and electric vehicle charging). It is also envisaged that electricity storage will become available and feasible for domestic use.

Various approaches have been proposed to determine which actions should take place and at what time they have to be scheduled. Some focus on optimising loads at a household level. They react to the different tariff bands by shifting flexible load to times when energy is cheaper, thus minimising the electricity bill for the consumer. Some also take into account on-site renewable generation and electricity storage facilities. This approach is most adequate in scenarios where indirect demand-side management is in place, mainly through time-of-use tariffs, since they optimise on behalf of the consumer. These household demand management techniques include scheduling based on historical trends (Rowe et al., 2013), using model predictive control on the environmental characteristics of each household (Yu et al., 2012), and constraint based mixed integer programming approaches (Scott et al., 2013).

Another point of view is that of the demand-side *aggregator*. This commercial entity imports power from the grid, through dealings on the energy market, which is then sold on to the consumers. As illustrated in Figure 7.1, any demand-side management mechanisms operating at this level have the advantage of seeing the bigger picture, and can thus balance between the loads of different households. One popular optimisation technique applied to this scenario also happens to be linear programming (Yixing Xu et al., 2010).

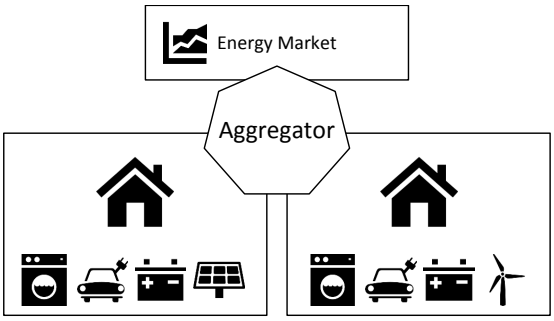


Figure 7.1: Interactions of a demand-side aggregator

Our focus is also on the demand-side aggregator. On an individual household level the options available are limited to a small subset of flexible loads, and energy prices are typically fixed to time-of-use tariffs. On the other hand, at an aggregate level, one can consider the flexible loads from all households, the feed-in tariffs of distributed generators, costs of using electricity storage facilities, and the unit cost prices set by the energy market. The temporal planning techniques proposed in this work can be used to model this domain and generate a plan

for a specific planning horizon. While this approach does not necessarily guarantee optimality, it can generate a feasible plan that satisfies the required constraints very quickly. It also has the advantage that unlike other approaches it does not require time discretisation, allowing the approach to scale gracefully for plans with a longer makespan.

7.2 A Formal Model for Aggregated Load

Let H be the set of households managed by a demand-side aggregator. For each household $h \in H$, we can split its consumption into inflexible load and flexible load. In this model, generation is assumed to be inflexible since it mostly depends on the weather, and cannot be shifted. It is also assumed that all the renewable energy generated is used or fed into the grid.

The function $p_h : \mathbb{R}_{>0} \to \mathbb{R}$ provides the inflexible instant power (in Watts) consumed by household h, at a specific point in time, while $g_h : \mathbb{R}_{>0} \to \mathbb{R}$ provides the instant power generated by household h, at a specific point in time. Equation 7.1 defines the instant power being consumed or generated by inflexible load, at an aggregate level, at a specific time $t \in \mathbb{R}_{>0}$.

$$p_{agg}(t) = \sum_{h \in H} p_h(t) - g_h(t) \tag{7.1}$$

7.2.1 Flexible Load

In a smart grid environment households might be ready to relinquish control of the time when certain activities should be performed, in return for some financial or ethical incentive. F_h defines the set of such flexible activities for household h. For each activity $f \in F_h$, $\alpha_f \in \mathbb{R}_{>0}$ is a variable that indicates its starting time. An activity also has a set of profiles Λ_f that can be used to perform the activity. For example, an electric vehicle might have two options to charge its battery, fast or slow, each of which has different duration and power requirements. The duration of an activity using a specific profile is provided by $dur_f : \Lambda_f \to \mathbb{R}_{>0}$.

We define the function $\Phi_f : \mathbb{R}_{>0} \times \Lambda_f \times \mathbb{R}_{>0} \to \mathbb{R}$, which takes the start time α_f of an activity f, the selected profile $\lambda_f \in \Lambda_f$, together with a specific time, t, and provides the load (in Watts) from the activity, at t. Equation 7.2 defines the aggregate instant power being consumed (or generated) by flexible and inflexible load, at a specific time t.

$$load_{agg}(t) = \sum_{h \in H} \left(p_h(t) - g_h(t) + \sum_{f \in F_h} \Phi_f(\alpha_f, \lambda_f, t) \right) \tag{7.2}$$

Each activity might have its constraints, which are specified in relation to its start time and duration. For instance, washing the dishes might have a user-defined constraint that $\alpha_f + dur_f(\lambda_f) < 0700hrs$, which indicates that the dishes must be ready before 7:00 a.m. Such constraints would be determined at an individual household level and communicated to the aggregator load control system via the smart grid's communication channels. Devices such as heating and cooling units can also determine such constraints autonomously, by monitoring the respective environmental characteristics and predicting the earliest and latest time heating or cooling needs to take place before the temperature goes out of the required trajectory bounds.

7.2.2 Electricity Storage

It is plausible that future aggregators would also have access to electricity storage facilities. These could be battery banks bought by consumers themselves (through schemes like the ones currently in place for investing in renewable energy generators) or by the aggregator, especially if the cost savings from shifting load are significant enough to justify the investment. Another possibility is to use storage capacity of batteries residing in electric vehicles (Kempton and Letendre, 1997). While this introduces a new level of uncertainty, since one cannot know when the user is going to disconnect the vehicle from the grid (to use it for transport), the same mechanism of deadline constraints described in the previous section can also be used to predict for how long the vehicle is likely to remain connected.

Electricity storage is not 100% efficient. Battery performance is subject to various factors such as internal resistance, discharge type (continuous versus intermittent), discharge load (constant load, constant current or constant power) and rate of charge/discharge, including overcharging and over-discharging (Chan and Sutanto, 2000). Battery age is also a contributing factor. A common way to approximate the state-of-charge (SOC) of a battery is through Peukert's law, which relates the available capacity of a battery to a charge or discharge rate, given a constant current charge or discharge. Peukert's law is often defined as $C_p = I^k t$, where C_p is the capacity in Ampere-hours (Ah), I is the discharge current, t is the actual time to discharge the battery in hours (h), and k is the Peukert constant, which varies depending on battery type and manufacturer. Batteries also have a self-discharge rate, which means that if a charged battery is left unused for a considerable amount of time it will still gradually lose its stored energy.

Let B be the set of batteries managed by the demand-side aggregator. The function $soc_b : \mathbb{R}_{>0} \rightarrow [0, 100]$ defines the state of charge of a battery $b \in B$, as a percentage of its total capacity, at a specific point in time. A battery can only be charged if its state of charge is lower than 100%, and it can only be discharged if its state of charge is greater than its minimum threshold, as recommended by the manufacturer. The function $p_b : \mathbb{R}_{>0} \rightarrow \mathbb{R}$ defines the power in or out, for battery b at a specific point in time. Equation 7.3 builds on Equation 7.2 to account for any load that is offset by the use of energy storage facilities.

$$load_{aggst}(t) = load_{agg}(t) + \sum_{b \in B} p_b(t) \tag{7.3}$$

7.2.3 The Aggregator's Objective Function

The demand-side aggregator could have various objectives. The most obvious one would be to minimise the costs required to supply the energy to its customers, thus maximising its profits. Another objective could be to minimise the peak-to-average ratio in order to maximise the lifetime of equipment and minimise maintenance costs. Alternatively, the aim of the aggregator could be to minimise the use of energy sources that use fossil fuels, which could also have financial eco-penalties associated with them.

Let $c : \mathbb{R} \times \mathbb{R}_{>0} \rightarrow \mathbb{R}$ be the function that determines the instantaneous cost for consuming energy from the grid (if the first argument is positive), or revenue from selling energy to the grid

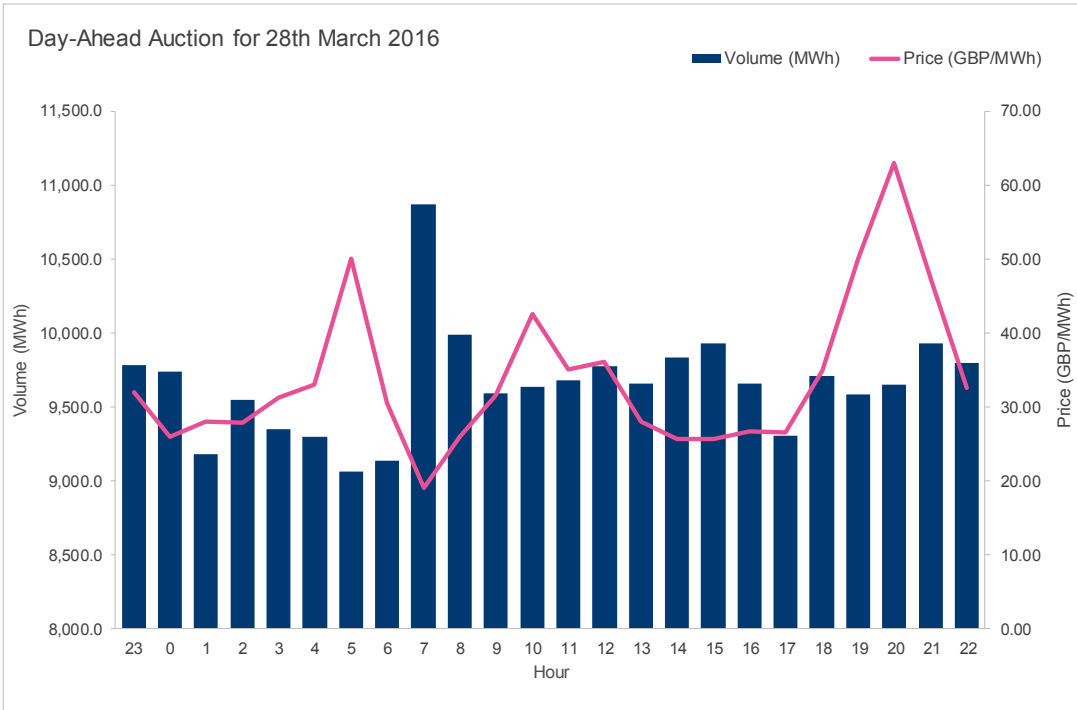


Figure 7.2: UK Day-Ahead Auction Results for 28th March 2016. *Source: APX Group (2016)*

(if the first argument is negative) at a specific point in time (the second argument). This may vary depending on the energy cost at that specific time. Figure 7.2 shows the results of the day-ahead auction of the UK energy market. The price per MWh at 20:00hrs is approximately double the price at 22:00hrs, so shifting load to a cheaper period can make a significant difference. Equation 7.4 defines the objective function m representing the aggregator's operational cost in terms of its cumulative load.

$$m = \int c(load_{aggst}(t), t)dt \tag{7.4}$$

The objective of the aggregator is to minimise m, using the variables that can be controlled. These are mainly the start time, α_f, and activity profile, λ_f, of each flexible activity, f, and the charge and discharge operations on each available battery, b.

7.3 A Piecewise Linear Temporal Planning Aggregator Model

As explained above, the load on the system is comprised of *inflexible* and *flexible* components. The inflexible component, such as renewable generation and inflexible demand, can only be predicted within a certain degree of accuracy from the information at hand (such as weather conditions, the type of day being a workday or a holiday, and any specific region-wide events that are predicted to cause spikes during specific times of the day, such as a football match or new year celebrations). On the other hand, flexible load provides direct choice as to when it is performed. Examples of such kind of activities include dish-washing, electric vehicle charging and also charging and discharging of storage within the network. Discharging produces negative flexible load that can help offset the effects of inflexible demand.

Modelling such a complex system in a planning domain is very difficult. However with the right simplifications that do not have a significant impact on the overall result, an adequate reasoning model can be designed such that a planning system that implements the techniques described so far can produce a solution. This can be validated with a higher fidelity simulator to evaluate its effectiveness and the error introduced from the respective simplifications.

7.3.1 Inflexible Demand and Generation

Since inflexible demand and generation cannot be influenced by the system their values can only be predicted up to a certain time using regression and time-series analysis on historical data. The inflexible load p_{agg}, defined in Equation 7.2 can thus be represented as a numeric function using TIFs (Piacentini et al., 2015), as described earlier in Section 3.4. Listing 7.1 shows an example of how the value of function `inflexible-load` can be changed, in this case every 15 time units, using TIFs. In this case, the `inflexible-load` is modelled as a step function.

```
(at 0  (= (inflexible-load) 400))
(at 15 (= (inflexible-load) 300))
(at 30 (= (inflexible-load) 320))
```

Listing 7.1: A Model of Inflexible Load as a Step Function using TIFs

```
(:objects g1 g2 - gradient)
(:init
  (= (current-time) 0)
  (= (inflexible-load) 400)
  (= (last-gradient-change-time) 0)
  (= (inflexible-load-dt) -6.667)
  (= (rate-of-change g1) 1.3333)
  (= (rate-of-change g2) 0)

  ;clip for timer envelope action
  (can-start-timer)
  (at 0.001 (not (can-start-timer)))

  ;clip for the first change in gradient
  (at 14.999 (can-change-rate g1))
  (at 15.001 (not (can-change-rate g1)))

  ;clip for the second change in gradient
  (at 29.999 (can-change-rate g2))
  (at 30.001 (not (can-change-rate g2))))

(:goal (and (changed-rate g1) (changed-rate g2)))
```

Listing 7.2: A Model of Inflexible Load with changes in Gradient ($\epsilon = 0.001$).

A better approach is to model the `inflexible-load` as a piece-wise linear function approximating the demand curve, with a numeric fluent representing the gradient between two points in time. This approach can offer a higher fidelity to the predicted demand curve but it is more complex to model, since an action needs to update the value of `inflexible-load` at each time point where the gradient changes. This can be achieved through tight clips (Fox

and Long, 2006; Coles and Coles, 2014), where a predicate is introduced at each time-point where a gradient change occurs, for a very small period of time, as described in Section 3.5.1. Goal conditions for each gradient change action are also introduced to force the updates to take place. For a planner that works with ϵ-separation, the tight clip for an action that is required to execute at time t has to start at $t - \epsilon$ and end at $t + \epsilon$, as shown in Listing 7.2. Since the `updateInflexibleLoad` action is mutex with the respective TIL it will be forced to be scheduled ϵ time after its precondition is satisfied, that is at time t.

The gradient for each segment is set through the numeric fluent `rate-of-change`, associated with a symbolic object for each gradient change. The actual switch is performed by an action `updateInflexibleLoad`, which calculates how much time elapsed since the last gradient change and updates the `inflexible-load` numeric fluent. The `current-time` numeric fluent keeps track of the actual time elapsed since the start of the plan. This is kept up to date by the `timer` envelope action (Coles et al., 2009a), which is clipped to start at the beginning of the time-line. This action has a continuous effect which increases the value of the `current-time` numeric fluent, with respect to time, `#t`. A record of the `last-gradient-change-time` is also maintained to be able to compute the new absolute value of the `inflexible-load` at the next gradient change linearly. Both actions are shown in Listing 7.3.

```
(:types gradient)
(:predicates
  (can-start-timer)
  (timer-used)
  (timing)
  (can-change-rate ?g - gradient)
  (changed-rate ?g - gradient))
(:functions
  (inflexible-load)
  (inflexible-load-dt)
  (last-gradient-change-time)
  (current-time)
  (rate-of-change ?g - gradient))

;update inflexible load
(:action updateInflexibleLoad
  :parameters (?g - gradient)
  :precondition (and (timing)
                     (can-change-rate ?g)
                     (not(changed-rate ?g)))

  :effect (and (increase (inflexible-load)
                         (* (- (current-time)
                               (last-gradient-change-time))
                            (inflexible-load-dt)))
               (assign (inflexible-load-dt)
                       (rate-of-change ?g))
               (assign (last-gradient-change-time)
                       (current-time))
               (changed-rate ?g)))
```

```
;envelope action to keep track of the time
(:durative-action timer
  :parameters()
  :duration (>= ?duration 0)
  :condition (and (at start (can-start-timer))
                  (at start (not (timing)))
                  (at start (not (timer-used))))
  :effect (and (at start (timing))
               (at start (timer-used))
               (at end (not (timing)))
               (increase (current-time) #t)))
```

Listing 7.3: Actions to Support Changing Gradients of Inflexible Load.

The same approach can be used to have a more granular model with inflexible generation separated from consumption, or separate components for the predicted generation and consumption of each household, if such information is available.

```
(:predicates
  (can-start-metering)
  (metered)
  (metering))
(:functions
  (inflexible-load)
  (inflexible-load-dt))

;envelope action to perform continuous metering
(:durative-action meter
  :parameters()
  :duration (>= ?duration 0)
  :condition (and (at start (can-start-metering))
                  (at start (not (metering)))
                  (at start (not (metered))))
  :effect (and (at start (metering))
               (at start (metered))
               (at end (not (metering)))
               (increase (inflexible-load)
                         (* #t (inflexible-load-dt)))))
```

Listing 7.4: A Model of Inflexible Load with changes in Gradient using TIFs.

```
(= (inflexible-load) 400)
(= (inflexible-load-dt) -6.667)

;clip for meter envelope action
(at 0 (can-start-metering))
(at 0.001 (not (can-start-metering)))

(at 15 (= (inflexible-load-dt) 1.33))
(at 30 (= (inflexible-load-dt) 0))
```

Listing 7.5: Problem Definition with changes to the Inflexible Load Gradient using TIFs.

With the availability of TIFs, this model can be simplified further, by setting the gradients directly instead of using clips. The envelope action will still be required, but its continuous

effect will update the `inflexible-load` with its rate of change `inflexible-load-dt`. This will produce the same piecewise linear result. As one can see from Listings 7.4 and 7.5 this construct makes modelling these kinds of effects much simpler.

For this model to work, the planner has to support piecewise linear continuous effects. To date, the only planner known to support this model is the hybrid planner UPMurphi (Della Penna et al., 2009), albeit very slow (and often runs out of memory before finding a solution). The technique described in Chapter 4 solves these kinds of models more efficiently.

7.3.2 Flexible Loads

As described earlier, flexible loads such as dish-washing, laundry, and electric vehicle charging can be scheduled such that they take place at an appropriate time when it is least impacting on the power network, and at the same time satisfies the constraints set by the user. Electric vehicles are of particular importance since they typically require a substantial amount of power for an extended period of time. As electric vehicles become more popular, new load peaks may start to appear during the night, when users of these vehicles would typically charge these vehicles. The decision variables for these flexible loads are typically the time at which the activity should take place and the selected charging activity profile. In our model, the flexible component of the total load is maintained separately by the `flexible-load` numeric fluent.

Each activity profile is associated with numeric fluents, such as the power needed and the duration required to perform the activity with that profile. This mechanism permits the attachment of any numeric values to an activity profile, thus modelling the set of profiles, Λ_f.

An activity can also have time constraints that determine when it is allowed to take place. The user might require an electric vehicle to be fully charged by 0700hrs and the dish washing to be done by 0600hrs, but not started before midnight. These constraints can be modelled using TILs, with a precondition predicate set to true at the time when the activity can take place and false otherwise. An activity can be modelled using a durative action (Fox and Long, 2003), with an invariant `overall` condition including the predicate set by the respective TIL.

Listing 7.6 shows the `perform` durative action, which is responsible for consuming the load associated with an activity's profile and marking the activity as `performed` at the end of the action. It checks that the activity has not yet been performed and that the activity profile is valid for that activity at the beginning of the action. It then verifies that the activity can be performed throughout the duration of the action, by checking the `can-perform` predicate for that activity. The effects of the `perform` durative action are that, at the beginning, the `flexible-load` is increased by the power needed, while at the end of the action, the `flexible-load` is decreased back by the same amount and the activity is marked as completed.

Listing 7.7 illustrates an example where two activity profiles `normal` and `fast` are associated with activity `charge-ev1-h1`, together with the respective power and duration needed to perform the activity with that profile. The activity can only take place between time 120 and time 600, controlled by `(can-perform charge-ev1-h1)`.

```
(:types activity profile)
(:predicates
   (activity-profile ?a - activity ?p - profile)
   (can-perform ?a - activity)
```

```
   (performed ?a - activity))
(:functions
   (flexible-load) ;flexible load
   (power-needed ?a - activity ?p - profile)
   (duration-needed ?a - activity ?p - profile))

(:durative-action perform
   :parameters (?a - activity ?p - profile)
   :duration (= ?duration (duration-needed ?a ?p))
   :condition (and
     (at start (not(performed ?a)))
     (at start (activity-profile ?a ?p))
     (over all (can-perform ?a)))
   :effect (and
     (at start (increase (flexible-load)
                         (power-needed ?a ?p)))
     (at end (decrease (flexible-load)
                       (power-needed ?a ?p)))
     (at end (performed ?a))))
```

Listing 7.6: A Model for Flexible Activities and Activity Profiles.

```
(:objects charge-ev1-h1 - activity
          normal fast - profile)
(:init
  (activity-profile charge-ev1-h1 normal)
  (= (power-needed charge-ev1-h1 normal) 3300)
  (= (duration-needed charge-ev1-h1 normal) 420)

  (activity-profile charge-ev1-h1 fast)
  (= (power-needed charge-ev1-h1 fast) 6600)
  (= (duration-needed charge-ev1-h1 fast) 260)

  (at 120 (can-perform charge-ev1-h1))
  (at 600 (not(can-perform charge-ev1-h1))))

(:goal (performed charge-ev1-h1))
```

Listing 7.7: Activity with its Activity Profiles and Allowed Time Window.

It is also possible that timing constraints between one action and another have to be in place. For instance, a clothes drying activity cannot take place before clothes washing has finished, or no two electric vehicles in the same house should be charged concurrently (to avoid overloading the circuit). This can be modelled trivially with a predicate that is an effect of the first action and a precondition of the second one. However, if constraints on the time between the two actions are in place a richer mechanism, such as strut actions (Fox et al., 2004) must be used, as described in Section 3.5.2.

7.3.3 Electricity Storage

In a similar fashion to flexible loads, electricity storage can also be modelled using charge and discharge profiles. These profiles would match the manufacturer specifications, possibly

following Peukert's Law, as described earlier. Listing 7.8 shows a charging and discharging profile, which is associated with storage facility `b1`, and specifies its operational characteristics in terms of wattage and energy stored in kilowatt-hours (kWh).

In order to support disconnections or unavailability of storage (such as in the case when the batteries of electric vehicles are used as storage) another predicate, `available`, has been added and can be controlled by other actions or through the use of TILs.

```
(:objects   b1 - battery
            normal - profile)
(:init
   (= (state-of-charge b1) 30)
   (available b1)

   (charge-profile b1 normal)
   (= (charge-power b1 normal) 2.0)
   (= (charge-rate b1 normal) 0.5)

   (discharge-profile b1 normal)
   (= (discharge-power b1 normal) 1.9)
   (= (discharge-rate b1 normal) 0.8)
```

Listing 7.8: Battery Charging and Discharging Profiles.

The `charge` and `discharge` operations are also modelled as durative actions, as shown in Listing 7.9. The precondition for the `charge` action states that the state-of-charge of the storage device, defined by the numeric fluent `state-of-charge`, is less than its maximum capacity, 100%, and that the battery is available during the whole charging cycle. The start and end effects of the `charge` durative action would be that the `flexible-load` is increased at the beginning of the action, to be decreased back at the end of the action. The `state-of-charge` associated with the storage device is increased gradually at the `charge-rate` associated with the chosen charging `profile` throughout the execution of the action. The `discharge` durative action does the exact opposite, depleting the `discharge-rate` gradually over time. The predicate `(in-use ?b - battery)` makes sure that the battery is not charged more than once concurrently, or charged and discharged at the same time. Furthermore, the duration of these actions is variable and the appropriate value needs to be chosen accordingly by the planning system, depending on the constraints and goals of the problem instance.

As discussed earlier, it is possible that the batteries in electric vehicles are used for storage (Kempton and Letendre, 1997), especially when there is surplus renewable generation which would otherwise be lost. In reality, it is difficult to predict when an electric vehicle is going to be available for such use and the only way this can be included in the planning problem is by inferring the time it is going to be available through the explicit user-specified constraints defining the deadline by which the electric vehicle needs to be charged. This means that the user is not expecting to need his electric vehicle before that time, and thus its battery can be used by the aggregator. Furthermore, batteries can also be charged using intermittent *smart charging* (Brooks et al., 2010). Rather than just selecting the best off-peak time when the battery should be charged, it involves a more fine-grained control with intermittent charging sessions.

```
(:types profile battery)
```

```
(:predicates
    (available ?b - battery)
    (in-use ?b - battery)
    (charge-profile ?b - battery ?p - profile)
    (discharge-profile ?b - battery ?p - profile))
(:functions
   (state-of-charge ?b - battery)
   (charge-power ?b - battery ?p - profile)
   (charge-rate ?b - battery ?p - profile)
   (discharge-power ?b - battery ?p - profile)
   (discharge-rate ?b - battery ?p - profile))

(:durative-action charge
  :parameters (?b - battery ?p - profile)
  :duration (>= ?duration 0)
  :condition (and
    (at start (not (in-use ?b)))
    (at start (charge-profile ?b ?p))
    (over all (<= (state-of-charge ?b) 100))
    (over all (available ?b)))
  :effect (and
    (at start (in-use ?b))
    (at start (increase (flexible-load)
                        (charge-power ?b ?p)))
    (at end (decrease (flexible-load)
                      (charge-power ?b ?p)))
    (at end (not (in-use ?b)))
    (increase (state-of-charge ?b)
              (* #t (charge-rate ?b ?p)))))

(:durative-action discharge
  :parameters (?b - battery ?p - profile)
  :duration (>= ?duration 0)
  :condition (and
    (at start (not (in-use ?b)))
    (at start (discharge-profile ?b ?p))
    (over all (>= (state-of-charge ?b) 0))
    (over all (available ?b)))
  :effect (and
    (at start (in-use ?b))
    (at start (decrease (flexible-load)
                        (discharge-power ?b ?p)))
    (at end (increase (flexible-load)
                      (discharge-power ?b ?p)))
    (at end (not (in-use ?b)))
    (decrease (state-of-charge ?b)
              (* #t (discharge-rate ?b ?p)))))
```

Listing 7.9: A Model for Electricity Storage Devices.

In this case the `charge-rate` and `discharge-rate` are assumed to be constant, and thus the `state-of-charge` follows a linear function. While this might be a good approximation for the purpose of modelling load levels, certain constraints might not be satisfied by a seemingly

compliant plan. This is especially the case if the linear approximation is too optimistic about the time required to charge an electric vehicle to the required level within a certain deadline.

7.3.4 Load Constraints

One of the objectives of the aggregator is to guarantee demand-side grid stability. This involves keeping operational parameters within certain bounds. A constraint on the total load can be easily introduced in our model, as an invariant condition on the sum of the `inflexible-load` and `flexible-load`. This can be added to the `meter` envelope action. A *flow tube* (Li and Williams, 2008) can be established by defining a maximum and minimum load level. It is also possible to have these values change over time with TIFs. This will not only ensure that operational limits on the load are respected, but also that sudden spikes and drops that can destabilise the power grid are avoided.

```
(over all (>= (+ (inflexible-load) (flexible-load))
                 (min-load)))

(over all (<= (+ (inflexible-load) (flexible-load))
                 (max-load)))
```

Listing 7.10: Bounds on the total load modelled as invariant conditions.

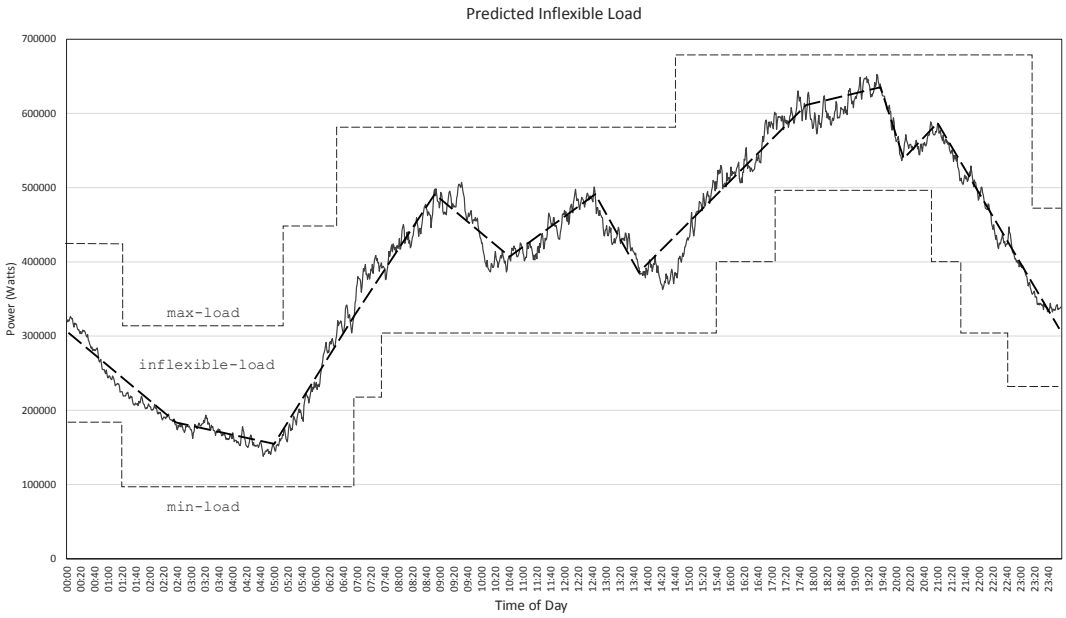


Figure 7.3: Predicted and Aproximated Inflexible Load with Load Thresholds.

Figure 7.3 illustrates these constraints overlaying the predicted inflexible load curve, which is piecewise linearly approximated with TIFs, specifying the local gradient of each linear segment. The maximum and minimum thresholds are also updated along the curve, creating a safe boundary that limits the planner from creating sharp peaks or drops from allocating excessive flexible load or discharging too much power from storage.

7.3.5 Energy Costs

A separate numeric fluent, `total-cost` is introduced, to account for the cost of energy consumed so far. At both aggregate level or consumer level, the unit cost can vary during the plan. At a consumer level it is already common to have different peak and off-peak tariffs since they serve as an incentive to influence the users' electricity consumption decisions. At an aggregate level, prices are set by the energy market, and typically change every hour. The price can be modelled as a separate numeric fluent, `unit-cost`, which can change at specific time-points during the plan, using TIFs, as shown in Listing 7.11.

```
(= (unit-price) 10)
(at 60 (= (unit-price) 1.2))
(at 120 (= (unit-price) 0.8))
(at 180 (= (unit-price) 0.5))
```

Listing 7.11: Updating the `unit-cost` with TIFs.

The `total-cost` can then be maintained up to date with a continuous effect on the same `meter` envelope action that wraps around the whole plan, clipped to start at the beginning of the plan. This continuous effect is defined in Listing 7.12.

```
(increase (total-cost)
          (* #t (* (unit-price)
                   (+ (flexible-load) (inflexible-load)))))
```

Listing 7.12: Non-linear Continuous Effect Calculating the Total Cost.

The problem with this expression is that it is non-linear and cannot be modelled as a linear program, since the `inflexible-load` is itself a piecewise linear function. If one assumes that the `unit-price` is constant for a specific period of time, this cost formulation can be simplified to exclude `inflexible-load`, and the `flexible-cost` can be computed as a piecewise linear function in terms of the `flexible-load`, as shown in Listing 7.13. This simplification is possible since the aggregator's control variables are anyway affecting only the `flexible-load`.

```
(increase (flexible-cost)
          (* #t (* (unit-price) (flexible-load))))
```

Listing 7.13: Piecewise Linear Continuous Effect Calculating the Flexible Cost.

This makes the planning problem solvable with the mechanism described in Chapter 4. This model has the disadvantage that constraints on the `total-cost` cannot be included, but it is still sufficient for a wide range of cases. Furthermore, this model does not support more complex cost models, such as banded tariffs, where the unit-cost depends on the amount of load consumed at that point in time. A more complex model that supports a more sophisticated cost model will be presented later on.

The full linear model, referred to as the Linear Aggregator domain, together with a sample problem instance are defined in Appendix D. The following section shows the results of some experiments with this domain.

7.4 Experiments with the Linear Aggregator Domain

The Linear Aggregator model was tested with the planner using the mechanism described in Chapter 4, referred to as DICE (Direct Interference of Continuous Effects). Listing 7.14 shows a sample solution plan for this domain.

```
0.0000: (meter) [820.0020]
9.9990: (charge b1 normal) [20.0000]
179.9990: (perform dishwasher1-h1 normal) [420.0000]
399.9990: (perform dishwasher1-h2 normal) [420.0000]
600.0010: (perform dishwasher1-h3 fast) [220.0000]
```

Listing 7.14: Sample plan for the Linear Aggregator domain.

For comparison it was also tested with UPMurphi (Della Penna et al., 2009), a hybrid planner. To date, no other planner that uses PDDL as input supports the numeric characteristics of this model. Both planners were executed on an Intel® Core™ i7-3770 CPU @ 3.40GHz. Both planners were allocated a maximum of 3GB of memory, due to the 32-bit limitation of UPMurphi.

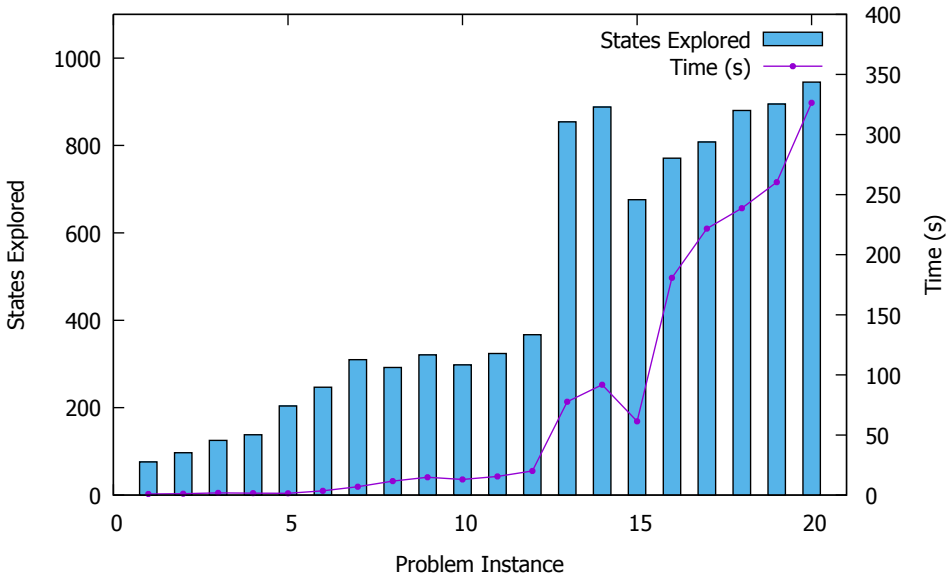


Figure 7.4: Performance on Linear Aggregator using EHC-ab with TRPGne.

The experiments were performed with problem instances of the Linear Aggregator domain described above. Each instance increases the level of complexity of the problem incrementally, by adding new activities that need to be performed with different activity profile choices, additional storage devices that need to be charged to a specific level, load bounds, and unit-price changes. All of these introduce more complexity to the problem and increase its branching factor exponentially. Figure 7.4 shows the performance of DICE on this problem set, showing the relationship between the time taken to solve the problem and the number of states explored.

Table 7.1 shows the results from these problem instances. It shows the number of units that needed to be scheduled within the constraints of the respective problem, and how many timed happenings (TH) were involved (which have an impact on the branching factor of the

Problem Instance				DICE			UPMurphi (10.0)	
#	Units	TH	Plan Steps	Time (s)	States	AB	Time	States
1	2	5	11	0.899	76	N	78.14	3,740,370
2	3	5	12	1.099	97	Y	1,437.38	61,313,989
3	4	6	16	1.782	125	Y	•	•
4	5	4	16	1.548	138	N	•	•
5	6	4	18	1.503	204	N	•	•
6	7	4	20	3.582	247	N	•	•
7	8	6	24	6.932	310	Y	•	•
8	9	12	32	11.623	292	Y	•	•
9	10	12	34	14.805	321	Y	•	•
10	11	13	37	12.971	298	N	•	•
11	12	13	39	15.554	324	N	•	•
12	13	14	42	20.103	367	N	•	•
13	14	14	44	77.631	854	Y	•	•
14	15	14	47	91.842	888	Y	•	•
15	16	15	49	61.362	676	Y	•	•
16	18	18	56	180.827	771	Y	•	•
17	19	18	58	221.753	808	Y	•	•
18	20	18	60	238.671	880	Y	•	•
19	21	18	62	260.346	895	Y	•	•
20	22	20	66	326.403	945	Y	•	•

Table 7.1: Experimental Results for Linear Aggregator Domain.

problem and the number of variables in each LP). The Plan Steps column indicates the number of discrete happenings in the plan. For the DICE planner, it shows the time taken in seconds, the number of states explored and whether the planner resorted to Ascent Backtracking (AB) due to EHC failing to find a plan. In the case of UPMurphi, the quantum had to be set to 10 time units to yield any result, and the initial clip for the `meter` action had to be widened to 20 time units due to this configuration, so the comparison is not completely like-with-like. This also means that in the case of UPMurphi, an action can only take place every 10th minute. Nevertheless, even with these simplifications, the difference between the two techniques on these kinds of problems is clear from these results.

As one can observe, the algorithm used by DICE is much more efficient when it comes to state exploration. The number of states explored is significantly lower than those of UPMurphi, which even at a very coarse granularity (10 time units) failed to find a plan for any but the smaller trivial problems. On the other hand, the time to evaluate each state in DICE is significant since for each state a linear program has to be computed multiple times (once to find a schedule, and twice for each numeric fluent to find their maximum and minimum feasible bounds), together with its corresponding TRPGne. The larger the number of discrete happenings in the plan, the more variables have to be modelled in each linear program, which also slows down its computation further. This has an impact on the algorithm's performance when EHC fails to find a solution and the planner has to resort to A*. EHC-ab alleviates some of this issue if a solution happens to be reachable from the state prior to the last hill-climb, as was the case in a lot of the problem instances of this domain.

Given the significant implementation differences, the time reported in the results should be taken into context, and only considered as an illustration of the difference in scalability due to the inherent nature of the algorithm used in DICE. Furthermore an implementation developed in a low-level language such as C++ or Go, that makes use of a commercial linear program solver such as CPlex or Gurobi, could deliver a better performance, especially when computing the lower and upper bounds of each numeric fluent needed as input to compute the heuristic value of each state.

7.5 Non-Linear Cost Aggregator

While the model described in Section 7.3 can suffice to generate an appropriate high level plan for simple costing models, it will not work for more complex energy pricing models that depend on banded tariffs, thus taking into account the total load of the system. It also does not account for the full total cost incurred by the aggregator, and thus cannot include constraints on such costs. The mechanisms proposed in Chapters 5 and 6 can be used to introduce a non-linear cost. Listing 7.15 shows the PDDLx class module definition for an aggregator with non-linear cost. The continuous function `total-cost-dt` encapsulates the non-linear rate of change of the `total-cost`, which can be used in the continuous effect expression of a durative action. The full PDDLx domain definition and a sample problem file are shown in Listings D.3 and D.4.

```
(define (module Kcl.Planning.APS.Aggregator)
  (:requirements :typing :fluents)
  (:functions
    (init (flexible-load))
    (init (inflexible-load))
    (mutable (inflexible-load-dt))
    (mutable (unit-price))
    (mutable (total-cost))
  (:continuous-functions
      (total-cost-dt)))
```

Listing 7.15: PDDLx Class Module Definition for the Non-Linear Cost Aggregator.

As discussed in Chapter 6, this process uses a non-linear iterative convergence mechanism to find the best linear approximation, within the specified precision, that fits the non-linear function. This can introduce significant overheads to the search procedure, which can impact scalability on large problem instances, as shown in Figure 7.5. This model was also tested with UPMurphi, using a quantum value of 10.0, using the non-linear continuous effect shown in Listing 7.12. UPMurphi only managed to solve the first two simple problem instances. In the case of uNICOrn, this was tested with a 1.0 precision for the non-linear iterative convergence algorithm together with the TRPGbr heuristic. The results are shown in Table 7.2.

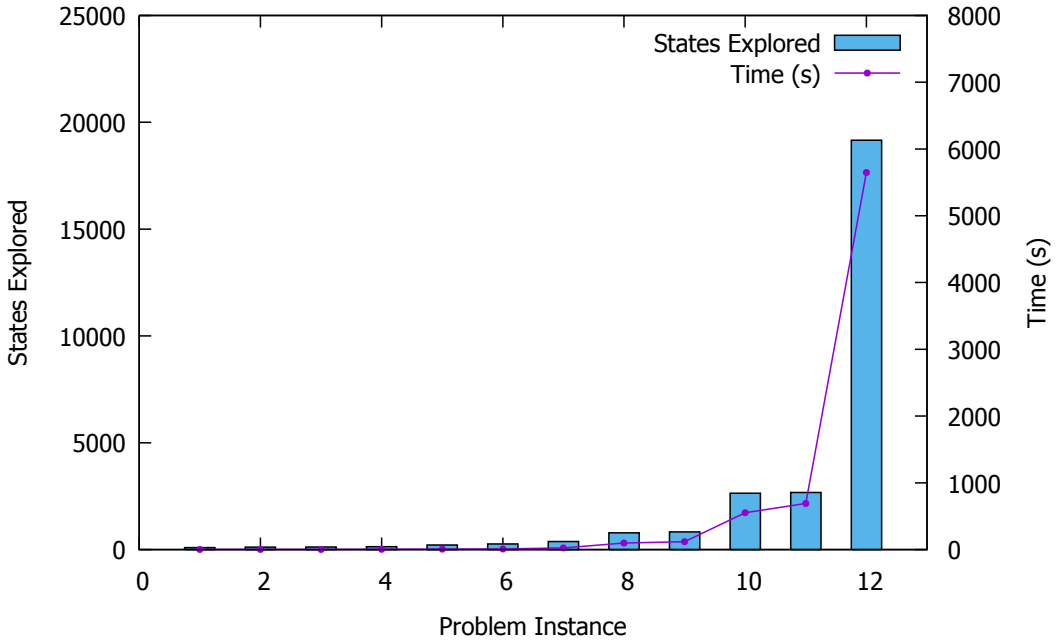


Figure 7.5: Performance on Non-Linear Cost Aggregator domain using EHC-ab with TRPGbr and Non-Linear Iterative Convergence with 1.0 Precision.

Problem Instance				uNICOrn (1.0) EHC-ab / TRPGbr			UPMurphi (10.0)	
#	Units	TH	Steps	Time (s)	States	AB	Time	States
1	2	6	12	1.445	92	N	88.62	4,346,720
2	3	8	16	2.388	112	Y	1,471.18	62,600,162
3	4	8	18	3.201	123	Y	•	•
4	5	8	20	3.790	139	N	•	•
5	6	7	21	6.312	212	N	•	•
6	7	7	23	9.133	265	N	•	•
7	8	9	27	24.332	379	Y	•	•
8	9	12	32	97.203	785	Y	•	•
9	10	12	34	117.742	827	Y	•	•
10	11	12	36	551.799	2,639	N	•	•
11	12	12	38	691.908	2,647	N	•	•
12	13	13	41	5,647.345	19,162	N	•	•

Table 7.2: Experimental Results for Non-Linear Cost Aggregator Domain.

7.6 Non-Linear Cost and Storage

An aggregator might also be interested in having a more accurate model of the storage systems within its network, such that it gets a more accurate prediction of their state of charge after a charging session. The state of charge of a battery discharges in a linear manner with constant power output, since it corresponds to the percentage of energy left in the battery out of the total capacity. However, as explained in Section 6.6.3, the increase in state of charge in relation to the time spent charging a battery typically follows a curve with diminishing returns.

```
(define (module Kcl.Planning.APS.Storage)
  (:requirements :typing :fluents)
  (:types Battery Profile)
  (:predicates
    (mutable (available ?b - Battery))
        (init (charge-profile ?b - Battery ?p - Profile))
        (init (discharge-profile ?b - Battery ?p - Profile)))
  (:functions
    (init (full-charge-time ?b - Battery))
    (init (charge-power ?b - Battery ?p - Profile))
    (init (discharge-power ?b - Battery ?p - Profile))
    (init (discharge-rate ?b - Battery ?p - Profile))
    (mutable (state-of-charge ?b - Battery)))
  (:continuous-functions
    (charge-rate ?b - Battery ?p - Profile)))
```

Listing 7.16: PDDLx Class Module Definition for Non-Linear Storage.

The external class module framework proposed can also be used to generate a plan that has a higher-fidelity prediction of the future state of the demand-side network where storage plays a more important role. The non-linear cost model was extended further with another external class module representing the non-linear continuous effect of charging a hypothetical battery. The charging curve used for the state of charge against time follows the Gauss error function $erf(t)$, scaled to match the typical duration of a domestic household or EV battery. The PDDLx definition of the `Storage` class module is shown in Listing 7.16.

This non-linear model acts as a good demonstration of how non-linear continuous behaviour can be incorporated in the planning problem with PDDLx external class modules. This functionality is not possible with any of the current state-of-the-art planning systems, including hybrid planners such as UPMurphi (Della Penna et al., 2009, 2012) or planners with semantic attachment mechanisms such TFD/M with PDDL/M (Dornhege et al., 2009). The performance of uNICOrn using 1.0 precision, with EHC-ab and TRPGbr is shown in Figure 7.6. The impact of non-linear iterative convergence when handling several non-linear variables becomes more evident as the number of explored states increases. The full PDDLx domain definition and a sample problem file are shown in Listings D.5 and D.6.

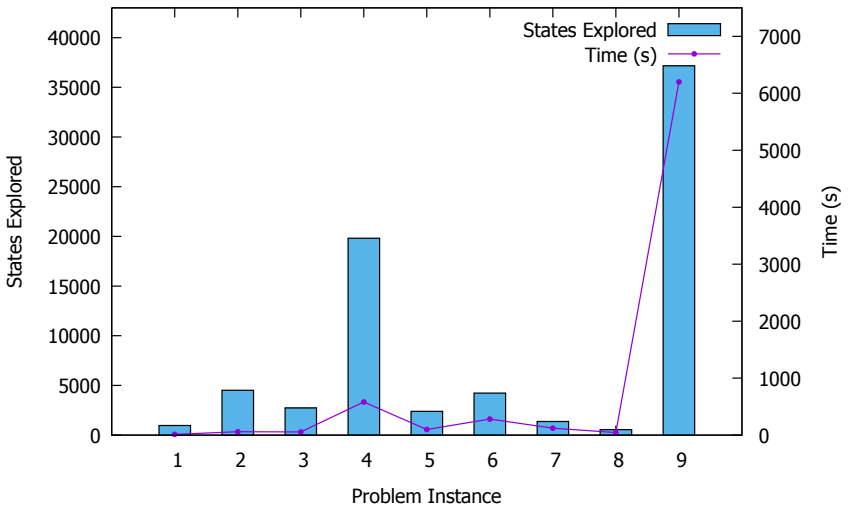


Figure 7.6: Performance on Non-Linear Cost and Storage Aggregator using EHC-ab with TRPGbr and Non-Linear Iterative Convergence with 1.0 Precision.

7.7 Automated Demand Dispatch Planning in the Real World

The ideas proposed in this work can be applied to solve temporal planning problems that feature the rich numeric characteristics, such as piecewise linear continuous effects due to constants in context and non-linear continuous effects that are monotonic within context. In order for such a planner to be used in the real-world further work needs to be done in three main areas, namely scalability, plan quality and uncertainty.

In the following sections we provide a few ideas about how these issues can be tackled such that the planner proposed in this work can be deployed in a real-world automated demand dispatch system.

7.7.1 Scalability

Planning is a hard problem, and even with the current state-of-the-art heuristics, scaling up to thousands of units is not feasible within acceptable times. However, aggregators responsible for demand dispatch need to support thousands of households. The following are some possible solutions to this problem that could be investigated.

The first solution is to perform incremental planning, starting from a small problem with a short planning horizon. This problem would only include activities that have a close deadline and thus have no other option than to be scheduled earlier in the plan. Once a feasible plan is found for the first set of more urgent activities, their load is absorbed into the `inflexible-load` component and a new plan for a longer planning horizon is computed. The new plan would include a new set of activities that have their deadline further away in time. Devices from this new set can still be scheduled early in the plan, if the operational constraints so demand, or if such a solution is more cost effective.

The second solution is to cluster devices that have similar deadlines and availability windows together in one virtual device. A lot of people have similar routines and work patterns, and this could be used to group similar segments together. For example, it is likely that whoever

has an electric vehicle will want to charge it overnight and have it ready by some time early in the morning. One could group a number of electric vehicles that have the same deadline, for instance at 8am, and cluster them into one schedulable unit. The selection of the size, in terms of cumulative load, of these units is a trade-off between getting a solution in an acceptable time and loss of granularity. However, in practice it is unlikely that an aggregator will be interested in single household level kilowatt granularity when managing a megawatt regional power network. The disadvantage of this approach is that the planner cannot take advantage of choices at a household level, such as selecting whether to charge a vehicle's battery quickly with high wattage versus slowly with a lower wattage. It is of course possible to create separate cluster units featuring the different charging profiles for the devices it represents, which the planner can choose from.

7.7.2 Plan Quality

The role of the aggregator is to satisfy demand of its customers while ensuring that the operational constraints are respected, such as balancing demand with supply and keeping the load within certain operational thresholds. However, it is also a commercial entity that purchases power from the energy market, and potentially also sells energy if the distributed low-voltage generators within its region, such as solar panels, generate any surplus. Given its predictions of demand and generation, the aggregator is not just interested in generating a plan that satisfies all the constraints. It is also interested in having a good quality plan, that minimises cost and potentially maximises revenue from surplus generation. For example, if the cost of energy is much higher in the evenings than during the day, the aggregator might opt to store as much as possible from any surplus energy generated from solar panels during the day (rather than selling it off directly) so that it is then used in the evening.

The techniques proposed in this work only generate solutions that satisfy the specified constraints of the problem, and are not guaranteed to provide an optimal plan. If the metric directive can be expressed linearly it can be used as the objective function of the linear program so that it minimises (or maximises) this metric rather than minimising makespan. However this mechanism only guarantees locally optimality, in the sense that it provides the optimal schedule given a fixed sequence of snap actions. For instance, if a durative action that represents an activity spans over a tariff change, the LP will move the durative action to make use of the cheaper segment as much as possible within the temporal constraints. Nevertheless, this does not mean that there is no other action sequence that can provide a even better quality plan in terms of the required objective function.

An anytime planning approach can be adopted to improve the quality of the plan. While it does not guarantee optimality it offers a trade off between computation time and plan quality. Anytime planning refers to a mechanism where the planning system provides a feasible plan as fast as possible, but then continues searching for better plans, until it is interrupted by the user. The advantage of this approach is that it can be used in situations where having a solution relatively quickly matters more than having an optimal plan, while if some more time is available a better quality plan could be found. After using the techniques proposed in this work to find the first feasible plan, it is possible to use one of the following solutions to improve its quality.

The first solution is to use the cost of the previous feasible plan as a constraint that is added automatically to the problem definition for the subsequent plan. So, if the generated solution has a `total-cost` of 10,000, a constraint is added to the `:goal` condition stating `(< (total-cost) 10000)`. This enforces the system to find a plan whose cost is lower than the previous one. The threshold can actually be lowered in steps if the user is not interested in finding plans that are just negligibly lower than the current one.

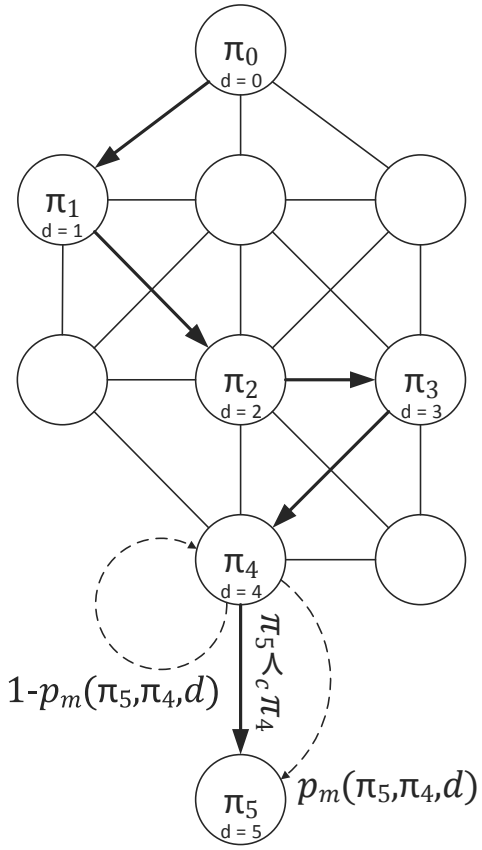


Figure 7.7: Local Search for a Better Quality Plan.

The second solution is to use local search, starting with the generated solution as the seed plan, π_0, to find a better quality solution. Local search works by generating a neighbourhood of solutions, $\mathcal{N}(\pi_0)$, with each solution in the neighbourhood corresponding to a modified version of π_0. These modifications typically include adding a new action, removing an action or moving an action to another position. If the resultant plan from such a modification, π_1 is a feasible valid plan, and $cost(\pi_1) < cost(\pi_0)$, it is returned as a better solution. Otherwise local search proceeds with exploring the neighbourhood of π_1 in the same way. As discussed in Section 2.9.5, local search is also susceptible to getting stuck in local minima. For this reason local search algorithms commonly include a stochastic element to randomly explore an inferior solution and escape the local minimum. Local search algorithms also typically include a distance threshold, d, such that if the local search explores too far without finding a solution it is interrupted and restarted.

Figure 7.7 illustrates this process, where $\pi' \prec_c \pi$ if $(violations(\pi') = 0) \wedge (violations(\pi) > 0 \vee cost(\pi') < cost(\pi))$, with *violations* representing the number of unsatisfied conditions in the

plan, introduced by the modification. p_m is an acceptance probability function that considers the characteristics of both the current and new solutions, together with the distance from the initial seed plan. Typically, the higher the distance, the lower the probability of accepting the new solution if it does not look particularly promising. LPG (Gerevini et al., 2003c), which also uses a random walk local search technique, also includes a heuristic evaluation as guidance to select the most promising plan from the neighbourhood.

7.7.3 Uncertainty

Thus far, it was always assumed that the planning agent has full visibility of the actions that need to be planned ahead, and that the predicted load levels are known beforehand with reasonable accuracy. In reality however, demand-side aggregators do not have such a luxury, especially when it comes to weather dependent predictions. Prediction accuracy is typically high for short planning horizons but drops significantly over longer periods. Furthermore, user requests are not received all at once, and new ones will be transmitted continually to the aggregator after a plan has been generated and is being executed. For this reason a *dynamic planning* solution is required, where the system reacts to dynamic changes in the environment.

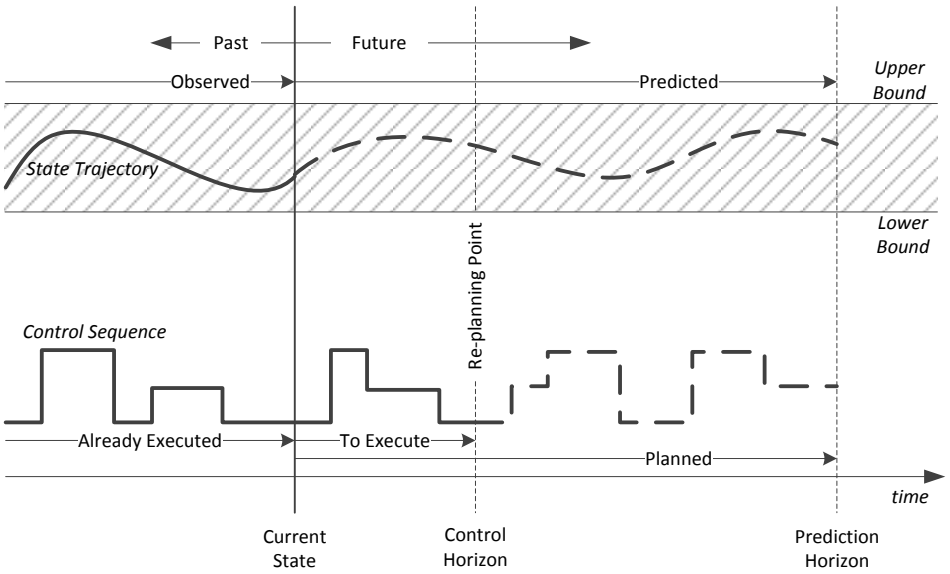


Figure 7.8: Receding Horizon Control

One possible solution to this is to adopt the idea used in Model Predictive Control (Rawlings, 2000), where the state trajectory of the system is predicted for a rolling time window starting from the current state, according to known exogenous events and planned control actions. This approach lends itself well to planning and has been used successfully in other domains that involve hybrid systems with numerical and logical state variables, where uncertainties due to changing environment information or system failures can occur (Löhr et al., 2013).

As illustrated in Figure 7.8, the idea is to only execute an initial segment of the whole plan, covering a shorter time window than what was originally planned. A new plan is then generated, which takes into account more up to date information and more accurate predictions. Since planning is computationally expensive, the time between each re-planning point needs to be

selected carefully. One could opt for regular re-planning intervals, but a more dynamic approach is also possible. Since the planner already computes the values of the numeric variables at each discrete time-point, it is possible to detect deviations from the predicted and actual trajectory at each control point, and trigger re-planning when this deviation exceeds a certain threshold. This approach avoids unnecessary computation if no unpredictable events occur.

A completely new plan could be generated from the current state, ignoring any planned activities, and reconsidering them for alternative schedules or execution profiles. Alternatively, a *plan repair* mechanism based on local search can also be employed to obtain a new plan. This would use the same plan improvement approach described above in Section 7.7.2, modifying the current solution incrementally until a new one is found, which satisfies the constraints.

7.8 Summary

In this chapter, the contributions of this work were applied to a real-world application, automated demand dispatch. This problem features a lot of the rich numeric characteristics that this research set out to tackle. This includes piecewise linear and also non-linear continuous functions.

A formal model of the problem was presented, based on existent literature in the domain of power systems and smart grids. This was then abstracted into a planning problem, which was modelled first in PDDL and then in PDDLx, to support non-linear continuous functions. This model accounts for background predicted inflexible load and generation, flexible activities with different fulfilment options, varying energy prices throughout the day representing different market rates, electricity storage which can be charged or discharged, and also global constraints on the load to avoid sudden peaks and drops which would destabilise the network.

Experimental results were presented, which demonstrate the applicability of this technology to solve problem instances of this domain. The results show that the proposed techniques are a viable enhancement to existent temporal numeric planners that handle linear continuous effects, such as COLIN (Coles et al., 2012) and POPF (Coles et al., 2010). These techniques can now handle a wider class of problems which is not solvable by any of the other planners, including LPG-td (Gerevini et al., 2004) and TFD (Eyerich et al., 2009). UPMurphi (Della Penna et al., 2009), a hybrid planner that can theoretically handle a much broader range of continuous functions, including non-linear ones, can only handle simple problem instances. The numeric and temporal characteristics of this model, mainly the constraints that need to be managed at a granular level and plan makespans that are hours long, make UPMurphi's approach, time discretisation, impractical to yield any useful results.

This chapter concludes with a proposal of how this technology can be incorporated into the architecture of a real automated demand dispatch system managed by a demand-side aggregator. Various issues were identified, such as how to scale beyond the planner's limitations, how to improve the quality of solutions, and how to handle uncertainty. Various ideas were proposed, which could themselves be avenues for further research.

8. Conclusion

The focus of this thesis is on temporal planning problems with rich numeric characteristics, especially those that require concurrency. The contributions of this work push the boundaries of temporal and numeric planning technology and open up several avenues for future work. The case study that inspired this research, the automated demand dispatch problem, introduced several interesting dynamics that necessitated the need for these new techniques. This chapter revisits the main contributions of this work and also provides direction for future research.

8.1 Summary of Main Contributions

The main objectives of this work were tof improve support for complex numeric behaviour in temporal planning. These were inspired from the automated demand dispatch problem faced by demand-side aggregators in power networks. Most of the work builds on existent temporal and numeric planning algorithms, including heuristic forward-chaining search, delete relaxation, and linear programming.

8.1.1 Planning with Constants in Context

The first contribution focuses on adding support for durative actions whose linear continuous effects are impacted by other instantaneous actions or timed state changes. This phenomenon is referred to as constants in context, where the rate of change of a continuous effect is only constant within a temporal context, that is the interval between two adjacent discrete happenings. A new planner, DICE, was developed, based on a new enhanced version of the algorithm used in COLIN (Coles et al., 2012). This new technique supports constants in context, widening the class of problems supported by DICE when compared to COLIN. A clear distinction was defined between time-dependent and non-time-dependent numeric variables, including the fact that a time-dependent variable at a step in a plan can become non-time-dependent again at a future step. Variables that are not time-dependent can have more complex numeric effects performed at discrete happenings, including non-linear functionality. This further enlarges the scope of supported planning problems.

The Temporal Relaxed Planning Graph (TRPG) originally introduced in CRIKEY3 (Coles et al., 2008) and also used in COLIN (Coles et al., 2012) was also enhanced to support these new dynamics. At each layer of the new numeric-enhanced Temporal Relaxed Planning Graph (TRPGne), the rate of change of each active continuous numeric effect is checked for any change in polarity, such that the bounds of the affected numeric variable are also increased or decreased

in the opposite direction. This effectively introduces support for cases where the piecewise linear function changes the sign of its gradient. The relaxed plan extraction process was also enhanced to support implicit preconditions. These are numeric conditions that are not explicitly stated by any action precondition, but are required for a specific numeric goal to be achieved. This was not necessary in COLIN because the rate of change of a continuous durative action was assumed to be constant, and thus no action could ever change this rate of change.

The popular Enforced Hill-Climbing (EHC) algorithm (Hoffmann and Nebel, 2001) was enhanced to perform better in the context of temporal planning with complex numeric constraints. Planners that use EHC, including Metric-FF (Hoffmann, 2003) and POPF (Coles et al., 2010) fall back to Weighted A*when EHC reaches a dead-end, often getting lost in the search and taking very long to yield a solution. The proposed Enforced Hill-Climbing with Ascent Backtracking (EHC-ab) algorithm keeps a stack of the hill-climbs performed by EHC. When EHC-ab reaches a dead-end, rather than starting an exhaustive search from the initial state, the state prior to the last hill-climb is popped off the stack and A*is performed from that state, effectively backtracking the last enforced hill-climb. Furthermore, only helpful actions are considered at this stage. This way, unhelpful states are pruned out, together with those that cannot reach the goal through the reachability analysis of the TRPGne.

If the search yields no results after backtracking the latest hill-climb a further hill-climb is backtracked from the stack. The process is repeated until no more states remain in the hill-climb stack, at which point the system will fall back to conventional A*. While this technique is still susceptible to get lost without yielding any result in a timely manner, it has proven to be very effective for the kinds of rich numeric domains analysed in this work.

8.1.2 A PDDL Extension for Semantic Attachment of External Modules

The second contribution focuses on providing support for computing complex numeric functions externally. Several approaches have been already investigated in the past, such as PDDL/M (Dornhege et al., 2009) and direct planner integration with an external solver (Piacentini et al., 2015). These are deemed to be too specific to the planner for which they were developed. The advantages and disadvantages of these approaches were analysed and the proposed approach was designed to keep as many of these advantages as possible, while addressing the disadvantages.

The end result of this effort was the PDDLx class module framework. This mechanism enhances the conventional PDDL language with a few extensions, reducing the barrier to entry for any planning system that is already capable of processing problems defined in PDDL 2.1 (Fox and Long, 2003). Unlike some of the prior approaches, this mechanism is planner and platform independent. The interface of an external class module is defined in PDDLx, and a planner supporting this framework would provide a tool that would process this definition and generate the necessary interface or library stubs for the module developer to implement. This process is completely transparent to the user of the planning system, and domains that make use of a class module can be moved between two different planning systems that provide their own implementation of the same class module definition.

PDDLx class module definitions allow the module designer to state which variables are immutable by planning actions and which of them can only be set through the problem's initial

state. This allows the planning system to validate the action schema with the module's declared interface. Class module methods also include hints as to which state variables are effected, which could be used by the planning system to deduce information which could be used in its search process. Furthermore, PDDLx was designed from a temporal planning perspective, including support for durations and continuous functions.

PDDLx was also designed to allow domain definitions to make use of multiple class modules. Modules are disambiguated using namespaces, and the state information pertaining to each module is protected from being accessed from other modules through a simple but effective type system. With this mechanism it is also possible to introduce a set of common external modules supported by all PDDLx planning systems, providing common functionality such as trigonometric functions or abstract data types.

8.1.3 Planning with Non-Linear Monotonic Continuous Functions

The third contribution incorporates non-linear monotonic continuous functions within the planning framework. This approach is based on finding the right linear approximations of the non-linear function, which is provided through the external class module framework. This technique is restricted to monotonic effects between discrete happenings, and only supports piecewise constant constraints where the cumulative effect on a numeric state variable is monotonic within a temporal context. While this is evidently more restrictive than more generic hybrid planning approaches such as those used in UPMurphi (Della Penna et al., 2009) and dReal (Bryce et al., 2015), it is significantly less computationally intensive, and still sufficient for a wide range of planning problems that feature non-linear continuous behaviour.

A non-linear iterative convergence algorithm was developed to find the right linear approximation for a non-linear continuous effect of a durative action with a flexible duration constraint. This algorithm improves its approximation iteratively until the error is within a certain error threshold, defined by the user.

The TRPGne heuristic was also adapted to include external functions provided through class modules. After observing the impact of having a large number of time-dependent non-linear numeric state variables a simpler heuristic was developed, the TRPGbr (backward-relaxed Temporal Relaxed Planning Graph). This only considers propositional variables and non-time-dependent numeric variables. This heuristic is not as informative as TRPGne but proves to be much faster to compute, which often compensates for any extra states that are explored. However, the effectiveness of this heuristic depends on the structure of the actions in the respective planning domain, and the non-time-dependent facts that they affirm in their effects.

8.1.4 Implementation and Evaluation

The aforementioned techniques were implemented in a new planner, developed in the Scala programming language. An initial evaluation for each technique was provided with some sample domains, such as the Project Planner domain and the Tanks domain, analysing the fundamental search metrics such as states explored and time with respect to the problem size. The system was then tested with the more realistic automated demand dispatch planning problem with the Linear and Non-Linear Aggregator domains. The characteristics of this domain were the

inspiration for this work and used as the case study for evaluating the applicability of the proposed techniques in a real world deployment.

8.2 Future Work

The contributions presented in this thesis open up several directions for further research, of both a theoretical and applied nature. In this section we describe some ideas that could be pursued in future work.

8.2.1 Support for Richer Non-Linear Characteristics

As described above, the non-linear iterative convergence technique described in this thesis is restricted to monotonic continuous functions. Furthermore, constraints on these functions are restricted to piecewise constant constraints. Further research in this area could introduce support for richer non-linear functionality.

The monotonicity limitation could be potentially alleviated by introducing auxiliary discrete states at known turning points of the non-linear function. This would require an extension to the internal interface between the planner and external class module, such that the system could enquire the timestamps of such turning points of the function, and introduce auxiliary happenings at those timestamps. Techniques such as piecewise linear approximations and convex hulls could also be investigated, in order to support more complex non-linear functions and constraints.

8.2.2 Further Heuristic Improvements

The heuristics presented in this work extend the popular delete-relaxation heuristic. However, one could not exclude the possibility of a more numeric-oriented heuristic which is more effective in providing search guidance in temporal problems with rich numeric constraints. If such a heuristic is discovered, it has the potential of significantly improving the scalability of current state-of-the-art temporal numeric planners.

8.2.3 Automated Heuristic Selection

In Section 6.4, the issue of selecting the most effective heuristic according to the characteristics of the problem was described. Some heuristics prove to be better suited than others for problems with certain characteristics. For example, TRPGne is more informative numerically than TRPGbr, because it computes the bounds of each numeric state variable, but is much more expensive to compute. The right balance between an informative but computationally expensive heuristic and a less informative but more efficient one seems to be problem dependent. Investigation into automated ways of determining which heuristic is best suited for a problem could lead to automatic selection of the right heuristic by the planning system rather than the user, making the planner even more domain independent.

8.2.4 Application to Control Parameters Extensions

While planning actions are typically grounded to symbolic discrete object arguments, there has been some renewed interest into *control parameters* (Fern et al., 2015) or *dynamics* (Li and Williams, 2008; Li and Williams, 2011): numeric arguments subject to some constraints. In these kinds of problems, the planner needs to determine the right feasible values to use such that all the constraints are satisfied and the goal is achieved. This introduces new dynamics to the planning process, and the contributions presented in this thesis can potentially be applied to these kinds of planning problems.

8.2.5 Real-World Deployment

As discussed in Section 7.7, there are several areas that need to be investigated further in order for planning technology to be deployed in the real-world. In the case of automated demand dispatch for electricity aggregators, the planning system needs to be integrated within a wider architecture. A model predictive control loop could be designed between the planner and the executive dispatch system, which uses new forecasts to plan for a receding horizon and perform automatic re-planning, thus handling uncertainty.

8.2.6 Plan Quality

While finding a plan that satisfies the constraints and requirements of a planning problem is critical in its own right, in reality the users often require a good quality plan. This means that the plan does not involve much redundant actions and the costs associated with the plan are as close to the optimal one as much as possible. Further investigation in this area is needed in order to provide good quality plans in an acceptable time-frame.

8.2.7 Scalability

Given the computational complexity of planning problems, even with the current state-of-the-art heuristics, scaling up to a high number of inputs is still a significant challenge. Future work in this area would definitely bring planning technology closer to being used in real-world problems.

8.2.8 Implementation in Other Planners

The techniques proposed in this work can be adapted, wholly and in part, to other planning systems. Planners that are based on forward search can have their state representation extended to include the necessary temporal state information. Each state evaluation can then be enhanced to solve a linear program for the simple temporal network for the plan to reach the current state from the initial one. The PDDL parser can be extended to support PDDLx and a semantic attachment mechanism can be introduced to interact with external class modules. The non-linear iterative convergence algorithm can subsequently be added to the system, based on the existent planning system machinery, to support non-linear monotonic functions. Planners based on local search can also be extended to include these techniques, although since the plans being

evaluated may not actually be valid (due to intermediate flaws), further research needs to be done in order to determine how such techniques can be adapted to such types of planners.

8.3 Final Remarks

A.I. Planning technology has made significant progress over the last couple of decades, and exciting research into temporal and numeric planning brings it closer to being used in real-world environments. The work presented in this thesis aims to be part of this contemporary effort to push the boundaries of this technology further.

The contributions of this work introduce several techniques to support planning problems that are not supported by most of the current state-of-the-art temporal and numeric planners. More specifically, this work introduces support for rich numeric characteristics such as piecewise linear numeric effects due to constants in context, non-linear monotonic continuous effects, and an extension framework to support the computation of complex functions through external class modules. A reference implementation of these techniques was developed and evaluated, proving their validity. These methods have opened up several avenues for future research, and one hopes that this work would be of inspiration to advance further the field of automated planning and artificial intelligence.

Appendices

A. PDDL for Problems with Constants in Context

This appendix contains the PDDL definitions of the domains evaluated in Chapter 4.

A.1 Project Planner

In this model the problem consists of tasks and resources. Only specific resources can perform certain tasks. Tasks require a specific duration to be completed. Tasks can also depend on other tasks to be completed first. A task with no dependencies is indicated with the `no-dependency` predicate, while a task which depends on another task to be completed before it is indicated with the `dependency` predicate. If a task depends on more than one task to be completed, a dummy `milestone` task can be introduced, which joins two tasks together with the `join-tasks` action that completes the dummy `milestone` task. This can then have a subsequent task which depends on it.

A `resource` is willing to work at specific hours of the day, and can have different rates at specific periods of the day. In the example shown in Listing A.2 each resource has a rate for the first 8 hours of the day, but a higher overtime rate for another 2 extra hours of the day. The goal is to `complete` all the tasks.

```
(define (domain project-planner)
  (:requirements
       :strips
       :typing
       :fluents
       :durative-actions
       :negative-preconditions
       :duration-inequalities
       :timed-initial-literals
       :timed-initial-fluents)

  (:types task resource)
  (:predicates
      (can-perform ?r - resource ?t - task)
      (occupied ?r - resource)
      (can-work ?r - resource)
      (complete ?t - task)
      (milestone ?t1 ?t2 ?m - task)
      (dependency ?pt ?t - task)
```

```
    (no-dependency ?t - task))

  (:functions
     (time-required ?r - resource ?t - task)
     (cost ?r - resource)
     (total-cost))

  ;perform a task with no dependencies
  (:durative-action perform-task
    :parameters (?r - resource ?t - task)
    :duration (= ?duration (time-required ?r ?t))
    :condition (and (at start (not (occupied ?r)))
                    (at start (not (complete ?t)))
                    (at start (no-dependency ?t))
                    (at start (can-perform ?r ?t))
                    (at start (can-work ?r))
                    (over all (can-work ?r)))
    :effect (and (at start (occupied ?r))
                 (at end (not (occupied ?r)))
                 (at end (complete ?t))
                 (increase (total-cost)
                           (* #t (cost ?r)))))

  ;perform a task if its dependency has been completed
  (:durative-action perform-dependent-task
    :parameters (?r - resource ?pt ?t - task)
    :duration (= ?duration (time-required ?r ?t))
    :condition (and (at start (not (occupied ?r)))
                    (at start (not (complete ?t)))
                    (at start (dependency ?pt ?t))
                    (at start (complete ?pt))
                    (at start (can-work ?r))
                    (at start (can-perform ?r ?t))
                    (over all (can-work ?r)))
    :effect (and (at start (occupied ?r))
                 (at end (not (occupied ?r)))
                 (at end (complete ?t))
                 (increase (total-cost)
                           (* #t (cost ?r)))))

  ;completes a "dummy" milestone task if two tasks are complete
  (:action join-tasks
    :parameters (?t1 ?t2 ?m - task)
    :precondition (and (milestone ?t1 ?t2 ?m)
                       (not (complete ?m))
                       (complete ?t1)
                       (complete ?t2))
    :effect (and (complete ?m)))
)
```

Listing A.1: PDDL Domain File for the Project Planner.

```
(define (problem project-planner-p3)
  (:domain project-planner)
  (:requirements
       :strips
       :typing
       :fluents
       :durative-actions
       :negative-preconditions
       :duration-inequalities
       :timed-initial-literals
       :timed-initial-fluents)

 (:objects task1 task2 task3 task4 task5 m1 m2 - task
           r1 r2 - resource)

 (:init
     (can-perform r1 task1)
     (= (time-required r1 task1) 3)

     (can-perform r1 task2)
     (= (time-required r1 task2) 5)

     (can-perform r2 task2)
     (= (time-required r2 task2) 3)

     (can-perform r1 task3)
     (= (time-required r1 task3) 6)

     (can-perform r2 task3)
     (= (time-required r2 task3) 4)

     (can-perform r2 task4)
     (= (time-required r2 task4) 2)

     (can-perform r1 task5)
     (= (time-required r1 task5) 5)

     (no-dependency task1)
     (no-dependency task2)
     (milestone task1 task2 m1)
     (dependency m1 task3)
     (no-dependency task4)
     (milestone task3 task4 m2)
     (dependency m2 task5)

     (= (total-cost) 0)
     (= (cost r1) 10)
     (= (cost r2) 12)

     (not (can-work r1))
     (not (can-work r2))
```

```
      (at 9 (can-work r1))
      (at 9 (can-work r2))

      (at 17 (= (cost r1) 15))
      (at 17 (= (cost r2) 18))

      (at 19 (not(can-work r1)))
      (at 19 (not(can-work r2)))

      (at 33 (= (cost r1) 10))
      (at 33 (= (cost r2) 12))
      (at 33 (can-work r1))
      (at 33 (can-work r2))

      (at 41 (= (cost r1) 15))
      (at 41 (= (cost r2) 18))

      (at 43 (not(can-work r1)))
      (at 43 (not(can-work r2)))
 )

 (:goal (and
    (complete task1)
    (complete task2)
    (complete task3)
    (complete task4)
    (complete task5)))

 (:metric minimize (total-cost))
)
```

Listing A.2: PDDL Problem File for the Project Planner.

A.2 Intelligent Pump Control

In this model, the problem consists of two types of industrial processes that require a flow of water to be present in their system. Pumps are installed to make sure the right water pressure is provided. `fill` processes need to fill a container, such as a boiler, up to a certain level. `use` processes use water continuously and flush it out, typically to keep the temperature of some equipment down as it is running. A process can depend on another process to be complete, and a process could also require another process to run concurrently with it. Those that do not have any dependencies will have the *constant* `ind-process plant` as their dependency, which is introduced permanently by the domain model. Each industrial process has a minimum `flow-rate` required to operate, and also a maximum excess flow rate, `max-excess-flow-rate`, which if exceeded would damage the equipment. Each `pump` has a minimum and maximum pump rate, and a `pump-flow-step`, by which the flow of the pump can be increased or decreased. A pump can be started by the `start-pump` action and stopped by the `stop-pump` action. The `increase-pump-flow` and `decrease-pump-flow` actions increase and decrease the pump's flow

rate respectively.

```
(define (domain pump-control)
    (:requirements
         :strips
         :typing
         :fluents
         :durative-actions
         :timed-initial-literals
         :negative-preconditions
         :duration-inequalities)

  (:types pump - object
          ind-process - object
          fill-process use-process - ind-process)

  (:constants plant - ind-process)

  (:predicates
     (dependency ?dep ?pr - ind-process)
     (conc-dependency ?dep ?pr - ind-process)
     (complete ?pr - ind-process)
     (running ?pr - ind-process))

  (:functions
    (current-flow-rate)
    (flow-rate ?pr - ind-process)
    (max-excess-flow-rate ?pr - ind-process)
    (duration-needed ?u - use-process)
    (min-fill-volume ?f - fill-process)
    (max-fill-volume ?f - fill-process)
    (current-volume ?f - fill-process)
    (current-pump-rate ?pu - pump)
    (min-pump-rate ?pu - pump)
    (max-pump-rate ?pu - pump)
    (pump-flow-step ?pu - pump))

  (:durative-action fill
    :parameters (?dep ?conc-dep - ind-process
                 ?f - fill-process)
    :duration (>= ?duration 0)
    :condition (and
      (at start (not(complete ?f)))
      (at start (not(running ?f)))
      (at start (dependency ?dep ?f))
      (at start (conc-dependency ?conc-dep ?f))
      (at start (complete ?dep))
      (at start (>= (current-flow-rate) (flow-rate ?f)))
      (over all (running ?conc-dep))
      (over all (<= (current-flow-rate)
                    (max-excess-flow-rate ?f)))
      (over all (<= (current-volume ?f) (max-fill-volume ?f)))
      (over all (>= (current-flow-rate) 0))
```

```
    (at end (>= (current-flow-rate) 0))
    (at end (>= (current-volume ?f) (min-fill-volume ?f))))
  :effect (and
    (at start (running ?f))
    (at start (decrease (current-flow-rate) (flow-rate ?f)))
    (increase (current-volume ?f)
              (* #t (+ (current-flow-rate) (flow-rate ?f))))
    (at end (increase (current-flow-rate) (flow-rate ?f)))
        (at end (not (running ?f)))
        (at end (complete ?f))))

(:durative-action use
  :parameters (?dep ?conc-dep - ind-process
               ?u - use-process)
  :duration (= ?duration (duration-needed ?u))
  :condition (and
    (at start (not(complete ?u)))
    (at start (not(running ?u)))
    (at start (dependency ?dep ?u))
    (at start (conc-dependency ?conc-dep ?u))
    (at start (complete ?dep))
    (at start (>= (current-flow-rate) (flow-rate ?u)))
    (over all (running ?conc-dep))
    (over all (<= (current-flow-rate)
              (max-excess-flow-rate ?u)))
    (over all (>= (current-flow-rate) 0))
    (at end (>= (current-flow-rate) 0)))
  :effect (and
    (at start (running ?u))
    (at start (decrease (current-flow-rate) (flow-rate ?u)))
    (at end (increase (current-flow-rate) (flow-rate ?u)))
    (at end (not (running ?u)))
    (at end (complete ?u))))

(:action start-pump
  :parameters (?p - pump)
  :precondition (and (= (current-pump-rate ?p) 0))
  :effect (and
    (assign (current-pump-rate ?p) (min-pump-rate ?p))
    (increase (current-flow-rate) (min-pump-rate ?p))))

(:action stop-pump
  :parameters (?p - pump)
  :precondition (and
    (< (current-pump-rate ?p) (+ (min-pump-rate ?p)
       (pump-flow-step ?p)))
    (> (current-pump-rate ?p) 0))
  :effect (and
    (decrease (current-flow-rate) (current-pump-rate ?p))
    (assign (current-pump-rate ?p) 0)))

(:action increase-pump-flow
```

```
    :parameters (?p - pump)
    :precondition (and
      (<= (current-pump-rate ?p) (+ (max-pump-rate ?p)
         (pump-flow-step ?p)))
      (>= (current-pump-rate ?p) (min-pump-rate ?p)))
    :effect (and
      (increase (current-pump-rate ?p) (pump-flow-step ?p))
      (increase (current-flow-rate) (pump-flow-step ?p))))

  (:action decrease-pump-flow
    :parameters (?p - pump)
    :precondition (and
      (>= (current-pump-rate ?p)
          (+ (min-pump-rate ?p) (pump-flow-step ?p))))
    :effect (and
      (decrease (current-pump-rate ?p) (pump-flow-step ?p))
      (decrease (current-flow-rate) (pump-flow-step ?p))))
)
```

Listing A.3: PDDL Domain File for the Intelligent Pump Control.

```
(define (problem pumpcontrol-p4)
  (:domain pumpcontrol)
  (:requirements
     :strips
     :typing
     :fluents
     :durative-actions
     :timed-initial-literals
     :negative-preconditions
     :duration-inequalities
     :timed-initial-fluents)
  (:objects
     p1 - pump
     f1 f2 f3 - fill-process
     u1 u2 - use-process)

  (:init
     (running plant)
     (complete plant)
     (= (current-flow-rate) 10)
     (= (current-pump-rate p1) 0)
     (= (min-pump-rate p1) 300)
     (= (max-pump-rate p1) 1200)
     (= (pump-flow-step p1) 100)

     (= (flow-rate f1) 100)
     (= (max-excess-flow-rate f1) 1000)
     (= (min-fill-volume f1) 30000)
     (= (max-fill-volume f1) 35000)
     (= (current-volume f1) 0)
     (dependency plant f1)
     (conc-dependency plant f1)
```

```
    (= (flow-rate f2) 150)
    (= (max-excess-flow-rate f2) 1500)
    (= (min-fill-volume f2) 20000)
    (= (max-fill-volume f2) 25000)
    (= (current-volume f2) 0)
    (dependency plant f2)
    (conc-dependency plant f2)

    (= (flow-rate f3) 150)
    (= (max-excess-flow-rate f3) 1500)
    (= (min-fill-volume f3) 30000)
    (= (max-fill-volume f3) 35000)
    (= (current-volume f3) 0)
    (dependency u1 f3)
    (conc-dependency plant f3)

    (= (flow-rate u1) 400)
    (= (max-excess-flow-rate u1) 1000)
    (= (duration-needed u1) 100)
    (dependency plant u1)
    (conc-dependency f1 u1)

    (= (flow-rate u2) 300)
    (= (max-excess-flow-rate u2) 1500)
    (= (duration-needed u2) 60)
    (dependency f2 u2)
    (conc-dependency plant u2))

    (:goal (and
      (complete f1)
      (complete f2)
      (complete f3)
      (complete u1)
      (complete u2)
      (= (current-pump-rate p1) 0)))
)
```

Listing A.4: PDDL Problem File for the Intelligent Pump Control.

A.3 Planetary Rover

In this model, a rover needs to autonomously plan its actions to perform a set of experiments and transmit the data with their results to the planetary orbiter, which will relay them back to Earth. Each action requires a certain amount of power, which is primarily supplied by the sun. `measure-power` is an envelope action clipped to start at the beginning of the plan, which keeps track of the changes in power levels received from solar energy. The `operate` action accounts for the basic operation of the rover's computer system and sensors. The `navigate` action takes the rover from one location to another. Experiments are performed using the `experiment` action if the rover is at the way point of the scientific objective. Both `navigate` and `experiment` rely

on solar power, and the plan needs to take into account the predicted solar power at the various times of the day. This is modelled through the `current-power` numeric fluent, whose rate of change is specified through the `current-solar-power-dt`, thus modelling solar power as a piecewise linear function. The planetary orbiter is scheduled to pass over specific waypoints at certain time intervals. The rover must be at the specific rendezvous to transmit data collected from experiments. Since this time window could coincide with periods when there is no solar power available, the `establish-uplink` and `transmit-experiment-data` actions rely on the rover's battery. This needs to be charged when solar power is available, with the `charge` action.

```
(define (domain planetary-rover)
  (:requirements
     :strips :typing :fluents :durative-actions
     :timed-initial-literals
     :timed-initial-fluents
     :negative-preconditions
     :duration-inequalities)
  (:types rover waypoint objective)
  (:predicates
     (can-start-measuring-power)
     (measuring-power ?r - rover)
     (measured-power ?r - rover)
     (operational ?r - rover)
     (rover-at ?r - rover ?w - waypoint)
     (can-travel ?from ?to - waypoint)
     (running-experiment ?o - objective)
     (objective-at ?o - objective ?w - waypoint)
     (complete ?o - objective)
     (travelling ?r - rover)
     (charging ?r - rover)
     (can-transmit-from ?w - waypoint)
     (transmitted-data ?o - objective)
     (uplink-established ?r - rover)
     (transmitting ?r - rover))
  (:functions
     (current-power ?r - rover)
     (power-needed-to-operate ?r - rover)
     (current-battery-power ?r - rover)
     (current-solar-power-dt ?r - rover)
     (power-needed-for-travel ?r - rover)
     (travel-time ?from ?to - waypoint)
     (experiment-duration ?r - rover ?o - objective)
     (power-needed-for-experiment ?r - rover ?o - objective)
     (power-needed-to-charge ?r - rover)
     (battery-charge-rate ?r - rover)
     (battery-capacity ?r - rover)
     (battery-energy ?r - rover)
     (battery-max-power ?r - rover)
     (payload-transmission-duration ?o - objective)
     (power-needed-for-transmission ?r - rover)
     (establish-uplink-energy ?r - rover)
     (teardown-uplink-energy ?r - rover))
```

```
(:durative-action measure-power
   :parameters (?r - rover)
   :duration (>= ?duration 0)
   :condition (and
     (at start (not (measuring-power ?r)))
     (at start (can-start-measuring-power)))
   :effect (and (at start (measuring-power ?r))
     (at start (measured-power ?r))
     (increase (current-power ?r)
               (* #t (current-solar-power-dt ?r)))
     (at end (not (measuring-power ?r)))))

(:durative-action operate
   :parameters (?r - rover)
   :duration (>= ?duration 0)
   :condition (and
    (at start (not(operational ?r)))
    (at start (>= (current-power ?r)
       (power-needed-to-operate ?r)))
    (at start (measuring-power ?r))
    (over all (measuring-power ?r))
    (at end (measuring-power ?r))
    (over all (>= (current-power ?r) 0)))
   :effect (and
    (at start (decrease (current-power ?r)
                        (power-needed-to-operate ?r)))
    (at start (operational ?r))
    (at end (increase (current-power ?r)
                      (power-needed-to-operate ?r)))
    (at end (not (operational ?r)))))

 (:durative-action navigate
   :parameters (?r - rover ?from ?to - waypoint)
   :duration (= ?duration (travel-time ?from ?to))
   :condition (and
     (at start (rover-at ?r ?from))
     (at start (can-travel ?from ?to))
     (at start (>= (current-power ?r)
                   (power-needed-for-travel ?r)))
     (at start (measuring-power ?r))
     (at start (not (travelling ?r)))
     (over all (measuring-power ?r))
     (at start (operational ?r))
     (over all (operational ?r))
     (over all (>= (current-power ?r) 0)))
 :effect (and
     (at start (decrease (current-power ?r)
                         (power-needed-for-travel ?r)))
     (at start (travelling ?r))
     (at start (not (rover-at ?r ?from)))
     (at end (rover-at ?r ?to))
```

```
    (at end (not (travelling ?r)))
    (at end (increase (current-power ?r)
                      (power-needed-for-travel ?r)))))

(:durative-action experiment
  :parameters (?r - rover ?w - waypoint ?o - objective)
  :duration (= ?duration (experiment-duration ?r ?o))
  :condition (and
    (at start (not (complete ?o)))
    (at start (not (running-experiment ?o)))
    (at start (objective-at ?o ?w))
    (at start (rover-at ?r ?w))
    (at start (operational ?r))
    (at start (>= (current-power ?r)
                  (power-needed-for-experiment ?r ?o)))
    (at start (measuring-power ?r))
    (over all (measuring-power ?r))
    (at end (measuring-power ?r))
    (over all (operational ?r))
    (over all (not (travelling ?r)))
    (over all (>= (current-power ?r) 0)))
  :effect (and
     (at start (running-experiment ?o))
     (at start (decrease (current-power ?r)
                         (power-needed-for-experiment ?r ?o)))
     (at end (increase (current-power ?r)
                       (power-needed-for-experiment ?r ?o)))
     (at end (not (running-experiment ?o)))
     (at end (complete ?o))))

(:durative-action establish-uplink
  :parameters (?r - rover ?w - waypoint)
  :duration (>= ?duration 0)
  :condition (and
    (at start (rover-at ?r ?w))
    (at start (can-transmit-from ?w))
    (at start (not (charging ?r)))
    (at start (not (uplink-established ?r)))
    (at start (>= (battery-max-power ?r)
                  (power-needed-for-transmission ?r)))
    (at start (>= (battery-energy ?r)
                  (+ (establish-uplink-energy ?r)
                     (teardown-uplink-energy ?r))))
    (at start (measuring-power ?r))
    (over all (measuring-power ?r))
    (at end (measuring-power ?r))
    (over all (not (travelling ?r)))
    (over all (>= (battery-max-power ?r) 0))
    (over all (>= (battery-energy ?r) 0)))
  :effect (and
     (at start (uplink-established ?r))
     (at start (decrease (battery-energy ?r)
```

```
                              (establish-uplink-energy ?r)))
        (decrease (battery-energy ?r)
                  (* #t (power-needed-for-transmission ?r)))
        (at end (decrease (battery-energy ?r)
                          (teardown-uplink-energy ?r)))
        (at end (not (uplink-established ?r)))))

    (:durative-action transmit-experiment-data
       :parameters (?r - rover ?o - objective)
       :duration (= ?duration
                    (payload-transmission-duration ?o))
       :condition (and
         (at start (not (transmitting ?r)))
         (at start (complete ?o))
         (at start (not (transmitted-data ?o)))
         (at start (measuring-power ?r))
         (over all (measuring-power ?r))
         (at end (measuring-power ?r))
         (at start (uplink-established ?r))
         (over all (uplink-established ?r)))
       :effect (and
         (at start (transmitting ?r))
         (at end (not (transmitting ?r)))
         (at end (transmitted-data ?o))))

    (:durative-action charge
      :parameters (?r - rover)
      :duration (>= ?duration 0)
      :condition (and
       (at start (>= (current-power ?r)
                     (power-needed-to-charge ?r)))
       (at start (< (battery-energy ?r)
                    (battery-capacity ?r)))
       (at start (not (charging ?r)))
       (at start (measuring-power ?r))
       (over all (measuring-power ?r))
       (at end (measuring-power ?r))
       (over all (>= (current-power ?r) 0))
       (over all (<= (battery-energy ?r)
                     (battery-capacity ?r))))
      :effect (and
       (at start (charging ?r))
       (at start (decrease (current-power ?r)
                           (power-needed-to-charge ?r)))
       (increase (battery-energy ?r)
                 (* #t (battery-charge-rate ?r)))
       (at end (increase (current-power ?r)
                         (power-needed-to-charge ?r)))
       (at end (not (charging ?r)))))
)
```

Listing A.5: PDDL Domain File for the Planetary Rover.

```
(define (problem planetary-rover-p2)
  (:domain planetary-rover)
  (:requirements
      :strips
      :typing
      :fluents
      :durative-actions
      :timed-initial-literals
      :negative-preconditions
      :duration-inequalities)

  (:objects
     rover1 - rover
     w0 w1 w2 - waypoint
     ob1 ob2 ob3 - objective)

  (:init
    (can-start-measuring-power rover1)
    (at 0.001 (not(can-start-measuring-power rover1)))

    (can-travel w0 w1)
    (= (travel-time w0 w1) 1)

    (can-travel w1 w2)
    (= (travel-time w1 w2) 2)

    (can-travel w2 w0)
    (= (travel-time w2 w0) 3)

    (at 18 (can-transmit-from w2))
    (at 21 (not (can-transmit-from w2)))

    (rover-at rover1 w0)

    (= (current-power rover1) 0)
    (= (current-solar-power-dt rover1) 0)
    (at 6 (= (current-solar-power-dt rover1) 5))
    (at 10 (= (current-solar-power-dt rover1) 3))
    (at 11 (= (current-solar-power-dt rover1) 0))
    (at 13 (= (current-solar-power-dt rover1) -3))
    (at 14 (= (current-solar-power-dt rover1) -5))
    (at 18 (= (current-solar-power-dt rover1) 0))

    (= (power-needed-to-operate rover1) 1)
    (= (power-needed-for-travel rover1) 0.2)

    (objective-at ob1 w1)
    (= (experiment-duration rover1 ob1) 2)
    (= (power-needed-for-experiment rover1 ob1) 2)
    (= (payload-transmission-duration ob1) 0.1)

    (objective-at ob2 w2)
```

```
    (= (experiment-duration rover1 ob2) 2)
    (= (power-needed-for-experiment rover1 ob2) 2)
    (= (payload-transmission-duration ob2) 0.1)

    (objective-at ob3 w2)
    (= (experiment-duration rover1 ob3) 1)
    (= (power-needed-for-experiment rover1 ob3) 3)
    (= (payload-transmission-duration ob3) 0.1)

    (= (power-needed-to-charge rover1) 3)
    (= (battery-charge-rate rover1) 2.2)
    (= (battery-capacity rover1) 300)
    (= (battery-max-power rover1) 25)
    (= (battery-energy rover1) 0)
    (= (power-needed-for-transmission rover1) 1)
    (= (establish-uplink-energy rover1) 2)
    (= (teardown-uplink-energy rover1) 1)
 )

 (:goal (and
     (transmitted-data ob1)
     (transmitted-data ob2)
     (transmitted-data ob3)))
)
```

Listing A.6: PDDL Problem File for the Planetary Rover.

B. BNF Description of PDDLx

PDDLx is an extension to the standard PDDL 2.1 (Fox and Long, 2003) and PDDL 2.2 (Edelkamp and Hoffmann, 2004) syntax, which is based on the original PDDL definition (McDermott et al., 1998). The BNF grammar below defines these extensions and should be considered as an addition to the standard PDDL BNF definition. Non-terminal tokens that are not defined below should be assumed to be identical to those defined in the PDDL BNF. Although not the focus of this work, the PDDLx extension is also compatible with the latest PDDL 3.1 grammar (Kovacs, 2011) and can thus be incorporated in a PDDL 3.1 compliant planning system.

As explained in Section 5.2 *class modules* are designed to support rich numeric properties, and inherently depend on numeric fluents and also types. For this reason `:numeric-fluents` and `:typing` have to be included in the **`:requirements`** section. Note that up till PDDL 2.2 `:numeric-fluents` was defined as `:fluents`, and they should be considered aliases of the same requirement.

B.1 Class Module Definition

A *class module definition* specifies the interface between the planner and the *class module*. It also provides the predicates, functions and methods available to be used in PDDLx domains that declare the *class module*.

⟨***module***⟩ ::= `(define (module` ⟨***qualified-name***⟩ `)`
[⟨***require-def***⟩]
[⟨***mod-types-def***⟩]:typing
[⟨***mod-constants-def***⟩]
[⟨***mod-predicates-def***⟩]
[⟨***mod-functions-def***⟩]:numeric-fluents
[⟨***cont-functions-def***⟩]:numeric-fluents
⟨***method-def***⟩* `)`

⟨***mod-types-def***⟩ ::=:typing `(:types` ⟨***mod-type-def***⟩+ `)`

⟨***mod-type-def***⟩ ::= ⟨***name***⟩ | ⟨***name***⟩+ – ⟨***name***⟩

⟨***mod-constants-def***⟩ ::= `(:constants` ⟨***mod-constant-def***⟩+ `)`

⟨***mod-constant-def***⟩ ::=:typing ⟨***name***⟩+ – ⟨***name***⟩

⟨***mod-predicates-def***⟩ ::= (:predicates ⟨***mod-predicate-def***⟩+)

⟨***mod-functions-def***⟩ ::=[:numeric-fluents] (:functions ⟨***mod-function-def***⟩+)

⟨***cont-functions-def***⟩ ::=[:numeric-fluents] (:continuous-functions
⟨***mod-function***⟩+)

⟨***mod-predicate-def***⟩ ::=[:typing] ⟨***mod-predicate***⟩
| (init | mutable ⟨***mod-predicate***⟩)

⟨***mod-function-def***⟩ ::=[:typing] ⟨***mod-function***⟩
| (init | mutable ⟨***mod-function***⟩)

⟨***typed-variables***⟩ ::= ⟨***name***⟩+ - ⟨***type***⟩

⟨***method-def***⟩ ::= (:method ⟨***method-symbol***⟩
:parameters (⟨***method-typed-args***⟩*)
:effects (⟨***method-effect***⟩*))

⟨***method-typed-args***⟩ ::= ⟨***name***⟩+ - ⟨***method-arg-type***⟩

⟨***method-arg-vars***⟩ ::= ⟨***name***⟩+ - ⟨***type***⟩

⟨***method-arg-type***⟩ ::= ⟨***type***⟩ | ⟨***primitive-arg-type***⟩

⟨***primitive-arg-type***⟩ ::= Boolean | Number

⟨***method-effect***⟩ ::= ⟨***mod-predicate***⟩
| (not | change ⟨***mod-predicate***⟩)
| ⟨***mod-function***⟩
| (increase | decrease | change ⟨***mod-function***⟩)

⟨***mod-predicate***⟩ ::= (⟨***predicate***⟩ ⟨***typed-variables***⟩*)

⟨***mod-function***⟩ ::= (⟨***function-symbol***⟩ ⟨***typed-variables***⟩*)

B.2 Domain Definition

A PDDLx domain that makes use of *class modules* will need to include the `:class-modules` requirement in the **`:requirements`** section. This is processed by planning systems supporting this requirement and the associated class modules will be loaded, together with any semantic attachment of *member predicates*, *functions* or execution of *methods* within action effects.

⟨***domain***⟩ ::= (define (domain ⟨***name***⟩)
[⟨***require-def***⟩]
[⟨***class-modules-def***⟩]
[⟨***types-def***⟩][:typing]
[⟨***constants-def***⟩]
[⟨***predicates-def***⟩]

[⟨***functions-def***⟩]:numeric-fluents
⟨***structure-def***⟩*)

⟨***class-modules-def***⟩ ::= (:classes ⟨***class-alias***⟩+)

⟨***class-alias***⟩ ::= ⟨***alias***⟩ - ⟨***qualified-name***⟩

⟨***alias***⟩ ::= ⟨***name***⟩

⟨***types-def***⟩ ::=:typing (:types ⟨***typed-list***⟩+)

⟨***constants-def***⟩ ::= (:constants ⟨***constant-def***⟩+)

⟨***constant-def***⟩ ::= ⟨***name***⟩

⟨***constant-def***⟩ ::=:typing ⟨***name***⟩+ - ⟨***m-type***⟩

⟨***predicates-def***⟩ ::= (:predicates ⟨***atomic-formula-skeleton***⟩+)

⟨***functions-def***⟩ ::=:numeric-fluents (:functions ⟨***atomic-formula-skeleton***⟩+)

⟨***atomic-formula-skeleton***⟩ ::= ⟨***predicate***⟩ ⟨***typed-variable-list***⟩+

⟨***typed-variable-list***⟩ ::= ⟨***variable***⟩*

⟨***typed-variable-list***⟩ ::=:typing ⟨***variable***⟩+ - ⟨***m-type***⟩

⟨***structure-def***⟩ ::= ⟨***action-def***⟩

⟨***structure-def***⟩ ::=:derived-predicates ⟨***derived-def***⟩

⟨***structure-def***⟩ ::=:durative-actions ⟨***durative-action-def***⟩

⟨***derived-def***⟩ ::= (:derived ⟨***atomic-formula-skeleton***⟩ ⟨***GD***⟩)

⟨***action-def***⟩ ::= (:action ⟨***action-symbol***⟩
:parameters (⟨***typed-variable-list***⟩+) ⟨***action-def-body***⟩)

⟨***action-symbol***⟩ ::= ⟨***name***⟩

⟨***action-def-body***⟩ ::= [:precondition ⟨***precondition-def***⟩]
[:effect ⟨***effect-def***⟩]

⟨***precondition-def***⟩ ::= () | ⟨***pre-GD***⟩

⟨***effect-def***⟩ ::= () | ⟨***effect***⟩

⟨***effect***⟩ ::= ⟨***c-effect***⟩ | (and ⟨***c-effect***⟩+)

⟨***c-effect***⟩ ::=:conditional-effects (forall (⟨***typed-variable-list***⟩+) ⟨***effect***⟩)
| (when ⟨***GD***⟩ ⟨***cond-effect***⟩)

⟨***c-effect***⟩ ::= ⟨***p-effect***⟩

⟨***p-effect***⟩ ::= (not ⟨***atomic-formula-term***⟩) | ⟨***atomic-formula-term***⟩

⟨***p-effect***⟩ ::=[:numeric-fluents] (⟨***assign-op***⟩ ⟨***f-head***⟩ ⟨***f-exp***⟩)

⟨***p-effect***⟩ ::=[:class-modules] ⟨***method-effect***⟩

⟨***cond-effect***⟩ ::= (and ⟨***p-effect***⟩*) | ⟨***p-effect***⟩

⟨***method-effect***⟩ ::= ⟨***alias***⟩.⟨***method***⟩

⟨***method***⟩ ::= ⟨***name***⟩

⟨***assign-op***⟩ ::= assign | scale-up | scale-down | increase | decrease

⟨***atomic-formula-term***⟩ ::= (⟨***m-predicate***⟩ ⟨***term***⟩*)

⟨***atomic-formula-term***⟩ ::=[:equality] (= ⟨***term***⟩ ⟨***term***⟩)

⟨***durative-action-def***⟩ ::= (:durative-action ⟨***da-symbol***⟩)
:parameters (⟨***typed-variable-list***⟩+)
⟨***da-def-body***⟩)

⟨***da-symbol***⟩ ::= ⟨***name***⟩

⟨***da-def-body***⟩ ::= :duration ⟨***duration-const***⟩
:condition ⟨***da-condition-def***⟩
:effect ⟨***da-effect-def***⟩

⟨***da-condition-def***⟩ ::= () | ⟨***da-GD***⟩

⟨***da-effect-def***⟩ = () | ⟨***da-effect***⟩

⟨***da-GD***⟩ ::= (and ⟨***da-GD***⟩*) | ⟨***timed-GD***⟩

⟨***timed-GD***⟩ ::= (at ⟨***time-specifier***⟩ ⟨***GD***⟩)
| (over all ⟨***GD***⟩)

⟨***time-specifier***⟩ ::= start | end

⟨***duration-const***⟩ ::= () | ⟨***simple-duration-const***⟩

⟨***duration-const***⟩ ::=[:duration-inequalities] (and ⟨***simple-duration-const***⟩+)

⟨***simple-duration-const***⟩ ::= (⟨***d-op***⟩ ?duration ⟨***d-value***⟩)
| (at ⟨***time-specifier***⟩ ⟨***simple-duration-const***⟩)

⟨***d-op***⟩ ::= =

⟨***d-op***⟩ ::=[:duration-inequalities] <= | >=

⟨***d-value***⟩ ::= ⟨***number***⟩

⟨***d-value***⟩ ::=[:numeric-fluents] ⟨***f-exp***⟩

⟨***da-effect***⟩ :: (and ⟨***da-effect***⟩*) | ⟨***timed-effect***⟩

⟨***da-effect***⟩ ::$^{\text{:conditional-effects}}$ (forall (⟨***typed-variable-list***⟩+) ⟨***da-effect***⟩)
| (when ⟨***da-GD***⟩ ⟨***timed-effect***⟩)

⟨***timed-effect***⟩ ::= (at ⟨***time-specifier***⟩ ⟨***cond-effect***⟩)

⟨***timed-effect***⟩ ::=$^{\text{:numeric-fluents}}$ (at ⟨***time-specifier***⟩ ⟨***f-assign-da***⟩)

⟨***timed-effect***⟩ ::=$^{\text{:continuous-effects}}$ (⟨***assign-op-t***⟩ ⟨***f-head***⟩ ⟨***f-exp-t***⟩)

⟨***f-assign-da***⟩ ::= (⟨***assign-op***⟩ ⟨***f-head***⟩ ⟨***f-exp-da***⟩)

⟨***f-exp-da***⟩ ::= (⟨***binary-op***⟩ ⟨***f-exp-da***⟩ ⟨***f-exp-da***⟩)
| (⟨***multi-op***⟩ ⟨***f-exp-da***⟩ ⟨***f-exp-da***⟩+)
| (- ⟨***f-exp-da***⟩)
| ⟨***f-exp***⟩

⟨***f-exp-da***⟩ ::=$^{\text{:duration-inequalities}}$?duration

⟨***assign-op-t***⟩ ::= increase | decrease

⟨***f-exp-t***⟩ ::= (* ⟨***f-exp***⟩ #t) | (* #t ⟨***f-exp***⟩) | #t

B.3 Problem Definition

A PDDLx problem definition follows suit from the respective domain it is associated with. The only difference between a PDDLx problem definition and a conventional PDDL one is that the objects defined in the problem can take types that were imported from a *class module* by the corresponding PDDLx domain. Furthermore, any specific predicates or functions marked with the **init** or **mutable** keywords in the *class module definition* can be initialised in the corresponding **:init** block.

⟨***domain***⟩ ::= (define (problem ⟨***name***⟩)
[⟨***require-def***⟩]
[⟨***objects-def***⟩]
[⟨***init***⟩]
[⟨***goal***⟩]
[⟨***metric-spec***⟩]$^{\text{:numeric-fluents}}$)

⟨***objects-def***⟩ ::= (:objects ⟨***typed-list***⟩+)

⟨***init***⟩ ::= (:init ⟨***init-el***⟩)

⟨***init-el***⟩ ::= ⟨***literal-name***⟩

⟨***init-el***⟩ ::=$^{\text{:numeric-fluents}}$ (= ⟨***basic-function-term***⟩ ⟨***number***⟩)

⟨***init-el***⟩ ::=$^{\text{:timed-initial-literals}}$ (at ⟨***number***⟩ ⟨***literal-name***⟩)

⟨***init-el***⟩ ::=$^{\text{:timed-initial-fluents}}$ (at ⟨***number***⟩ (= ⟨***basic-function-term***⟩ ⟨***number***⟩))

⟨***basic-function-term***⟩ ::= ⟨***m-function-symbol***⟩
| (⟨***m-function-symbol***⟩ ⟨***name***⟩*)

⟨***literal-name***⟩ ::= ⟨***atomic-formula-name***⟩
| (not ⟨***atomic-formula-name***⟩)

⟨***atomic-formula-name***⟩ ::= (⟨***m-predicate***⟩ ⟨***name***⟩*)

⟨***atomic-formula-name***⟩ ::=$^{\text{:equality}}$ (= ⟨***name***⟩ ⟨***name***⟩)

⟨***goal***⟩ ::= (:goal ⟨***pre-GD***⟩)

⟨***metric-spec***⟩ ::=$^{\text{:numeric-fluents}}$ (:metric ⟨***optimization***⟩ ⟨***metric-f-exp***⟩)

⟨***optimization***⟩ ::= minimize|maximize

⟨***metric-f-exp***⟩ ::= (⟨***binary-op***⟩ ⟨***metric-f-exp***⟩ ⟨***metric-f-exp***⟩)
| (⟨***multi-op***⟩ ⟨***metric-f-exp***⟩ ⟨***metric-f-exp***⟩+)
| (- ⟨***metric-f-exp***⟩)
| ⟨***number***⟩
| (⟨***m-function-symbol***⟩ ⟨***name***⟩*)
| ⟨***m-function-symbol***⟩
| total-time

B.4 Common Definitions

⟨***qualified-name***⟩ ::= [⟨***name***⟩.]+⟨***name***⟩

⟨***require-def***⟩ ::= (:requirements ⟨***require-key***⟩)

⟨***require-key***⟩ ::= See Appendix B.5.

⟨***pre-GD***⟩ ::= ⟨***GD***⟩ | (and ⟨***pre-GD***⟩)

⟨***GD***⟩ ::= ⟨***atomic-formula-term***⟩ | (and ⟨***GD***⟩*)

⟨***GD***⟩ ::=$^{\text{:numeric-fluents}}$ ⟨***f-comp***⟩

⟨***GD***⟩ ::=$^{\text{:disjuntive-preconditions}}$ (or ⟨***GD***⟩*) | (not ⟨***GD***⟩) | (imply ⟨***GD***⟩ ⟨***GD***⟩)

⟨***GD***⟩ ::=$^{\text{:existential-preconditions}}$ (exists (⟨***typed-variable-list***⟩+) ⟨***GD***⟩)

⟨***GD***⟩ ::=$^{\text{:universal-preconditions}}$ (forall (⟨***typed-variable-list***⟩+) ⟨***GD***⟩)

⟨***f-comp***⟩ ::= (⟨***binary-comp***⟩ ⟨***f-exp***⟩ ⟨***f-exp***⟩)

⟨***f-exp***⟩ ::= ⟨***number***⟩
| (⟨***binary-op***⟩ ⟨***f-exp***⟩ ⟨***f-exp***⟩)
| (⟨***multi-op***⟩ ⟨***f-exp***⟩ ⟨***f-exp***⟩+)

| (- ⟨***f-exp***⟩)
| (⟨***f-head***⟩)

⟨***f-head***⟩ ::= (⟨***m-function-symbol***⟩ ⟨***term***⟩*)
| ⟨***m-function-symbol***⟩

⟨***term***⟩ ::= ⟨***name***⟩ | ⟨***variable***⟩

⟨***binary-op***⟩ ::= - | / | ⟨***multi-op***⟩

⟨***multi-op***⟩ ::= + | *

⟨***binary-comp***⟩ ::= > | < | = | >= | <=

⟨***typed-list***⟩ ::= ⟨***name***⟩*

⟨***typed-list***⟩ ::=$^{\text{:typing}}$ ⟨***name***⟩+ - ⟨***m-type***⟩

⟨***m-type***⟩ ::= ⟨***type***⟩

⟨***m-type***⟩ ::=$^{\text{:class-modules}}$ ⟨***alias***⟩.⟨***name***⟩

⟨***m-predicate***⟩ ::= ⟨***predicate***⟩

⟨***m-predicate***⟩ ::=$^{\text{:class-modules}}$ ⟨***alias***⟩.⟨***predicate***⟩

⟨***m-function-symbol***⟩ ::= ⟨***function-symbol***⟩

⟨***m-function-symbol***⟩ ::= $^{\text{:class-modules}}$ ⟨***alias***⟩.⟨***name***⟩

⟨***type***⟩ ::= ⟨***name***⟩ | (either ⟨***name***⟩+)

⟨***predicate***⟩ ::= ⟨***name***⟩

⟨***function-symbol***⟩ ::= ⟨***name***⟩

⟨***variable***⟩ ::= ?⟨***name***⟩

⟨***name***⟩ ::= ⟨***letter***⟩⟨***any-char***⟩*

⟨***any-char***⟩ ::= ⟨***letter***⟩ | ⟨***digit***⟩

⟨***number***⟩ ::= ⟨***digit***⟩+ [⟨***decimal***⟩]

⟨***decimal***⟩ ::= .⟨***digit***⟩+

⟨***letter***⟩ ::= a..z | A..Z

⟨***digit***⟩ ::= 0..9

B.5 PDDLx Requirements

Several features and extensions were added to PDDL over the years. A domain that makes use of a specific feature which is not part of the mandatory PDDL functionality (`:strips` classical planning) can specify it as a *requirement* in the **`:requirements`** definition. The planning system will process the list of *requirements* and compare them with the ones it supports. A domain that makes use of PDDLx external class modules will include the `:class-modules` requirement. In addition, required support for *timed initial fluents*, which initialise the value of a numeric fluent at a specific time (as described in Section 2.4.2), is indicated by `:timed-initial-fluents`.

For the latest list of additional possible requirements (at time of writing) refer to Section 1.3 of the PDDL 3.1 BNF definition (Kovacs, 2011).

C. PDDLx for Non-Linear Domains

This appendix contains the PDDL definitions of the domains evaluated in Chapter 6.

C.1 Tanks

In this simple tanks domain, a container, referred to as a `bucket`, needs to be filled up to a certain level. The rate at which the liquid inside a tank exits the spigot, and the rate at which the liquid level inside the tank decreases are both non-linear, approximated by Torricelli's Law. This behaviour is encapsulated inside an external *class module*. The PDDLx definition of the interface to this module is shown in Listing C.1. This module provides three parameters that can be initialised, the `height` of the liquid in the tank, the `radius` of the surface area of the liquid in the tank, and the `hole-radius` of the spigot from which the liquid exits. Furthermore, the `height` can be modified by the actions in the plan itself. The `drain-rate` and `height-change`, representing the rate at which the liquid exits the tank and the rate at which the height of the liquid in the tank changes respectively, are calculated by the module.

In order to test the behaviour with multiple tanks, a `filled-from` predicate was introduced, and the respective goals were added to force the system to use multiple tanks. This also helps the heuristic guidance to select helpful actions, allowing the non-linear iterative convergence part to be turned off when performing the heuristic evaluation of each successor state.

```
(define (module Kcl.Planning.Torricelli)
  (:requirements :typing :fluents)

  (:types Tank)

  (:functions
     ;the surface area of the liquid in the tank
     (init (radius ?t - Tank))
     ;the surface area of the hole in the tank
     (init (hole-radius ?t - Tank))
     ;the height of the liquid in the tank
     (mutable (height ?t - Tank)))

  (:continuous-functions
     (drain-rate ?t - Tank)
     (height-change ?t - Tank))
)
```

Listing C.1: PDDLx Definition for Torricelli Class Module.

```
(define (domain tanks)
  (:requirements
       :strips
       :typing
       :fluents
       :durative-actions
       :duration-inequalities
       :negative-preconditions
       :class-modules)

  (:classes Torricelli - Kcl.Planning.Torricelli)

  (:types Bucket)

  (:predicates
      (filling ?b - Bucket)
      (filled-from ?t - Torricelli.Tank ?b - Bucket)
      (filling-from ?t - Torricelli.Tank))

  (:functions
     ;how much water do we have in the bucket
     (volume ?b - Bucket)

     ;how much water can the bucket hold
     (capacity ?b - Bucket))

  (:durative-action fill
     :parameters (?t - Torricelli.Tank ?b - Bucket)
     :duration (and (>= ?duration 0))
     :condition (and (over all (>= (Torricelli.height ?t) 0))
                     (over all (<= (volume ?b) (capacity ?b)))
                     (at start (not(filling ?b)))
                     (at start (not(filled-from ?t ?b)))
                     (at start (not(filling-from ?t))))
     :effect (and
        (at start (filling ?b))
        (at start (filling-from ?t))
        (increase (volume ?b)
                  (* #t (Torricelli.drain-rate ?t)))
        (decrease (Torricelli.height ?t)
                  (* #t (Torricelli.height-change ?t)))
        (at end (not(filling ?b)))
        (at end (not(filling-from ?t)))
        (at end (filled-from ?t ?b))))
)
```

Listing C.2: PDDLx Definition for the Tanks domain.

```
(define (problem tanks-p03)
   (:domain tanks)
   (:requirements
      :strips
      :typing
```

```
        :fluents
        :durative-actions
        :duration-inequalities
        :negative-preconditions
        :class-modules)

    (:objects
        bucket1 - Bucket
        tank1 tank2 tank3 - Torricelli.Tank)

    (:init
        (= (Torricelli.height tank1) 10)
        (= (Torricelli.radius tank1) 5)
        (= (Torricelli.hole-radius tank1) 0.05)

        (= (Torricelli.height tank2) 10)
        (= (Torricelli.radius tank2) 5)
        (= (Torricelli.hole-radius tank2) 0.05)

        (= (Torricelli.height tank3) 10)
        (= (Torricelli.radius tank3) 5)
        (= (Torricelli.hole-radius tank3) 0.05)

        (= (volume bucket1) 0)
        (= (capacity bucket1) 2250))

    (:goal
        (and (> (volume bucket1) (- (capacity bucket1) 100))
             (<= (volume bucket1) (capacity bucket1))
             (filled-from tank1 bucket1)
             (filled-from tank2 bucket1)
             (filled-from tank3 bucket1)))
)
```

Listing C.3: PDDLx Problem Instance for the Tanks Domain.

C.2 Thermostat

This domain models a Bounded Trajectory Management Problem (BTMP) (Piacentini et al., 2015) representing an object whose temperature needs to be maintained within certain bounds throughout the required `planning-horizon`. Listing C.4 shows the definition of the `Kcl.Planning.Temperature` *class module*, which computes the `cooling-rate` according to Newton's Law of Cooling. It introduces a type `Thermal`, for which the `cooling-constant` and `current-temp` can be defined. If direct heat is being applied to the object, the `heating` predicate is set to *true*.

Listing C.5 defines the temperature bounds of each object being monitored, together with the `monitor` envelope action, which applies the `cooling-rate` computed from the class module, and the `heat` durative action, which applies the `heating-rate`. Listing C.6 shows an example of a problem instance for this domain, with a `planning-horizon` of 90, and initial ambient

temperature of -6, with predicted changes to the ambient temperature defined through timed initial fluents. The `monitor` action is *clipped* to start at the beginning of the plan with the timed initial literal `can-start-monitoring` set to *false* at timestamp 0.001.

```
(define (module Kcl.Planning.Temperature)
  (:requirements :typing :fluents)
  (:types Thermal)

  (:predicates
    (mutable (heating ?o - Thermal)))

  (:functions
    (init (ambient-temp))
    (init (current-temp ?o - Thermal))
    (init (cooling-constant ?o - Thermal)))

  (:continuous-functions
    (cooling-rate ?o - Thermal))
)
```

Listing C.4: PDDLx Definition for Temperature Class Module.

```
(define (domain thermostat)
    (:requirements
         :strips
         :typing
         :fluents
         :durative-actions
         :timed-initial-literals
         :timed-initial-fluents
         :negative-preconditions
         :duration-inequalities
         :class-modules)

   (:classes Temperature - Kcl.Planning.Temperature)

   (:predicates
       (Temperature.heating ?o - Temperature.Thermal)

       (monitoring ?o - Temperature.Thermal)
       (monitored ?o - Temperature.Thermal)
       (can-start-monitoring)
   )

   (:functions
      (Temperature.ambient-temp)
      (Temperature.current-temp ?o - Temperature.Thermal)
      (Temperature.cooling-constant ?o - Temperature.Thermal)

      (planning-horizon)
      (minimum-temp ?o - Temperature.Thermal)
      (maximum-temp ?o - Temperature.Thermal)
      (thermostat-on-temp ?o - Temporal.Thermal)
```

```
      (thermostat-minoff-temp ?o - Temporal.Thermal)

      (heating-rate ?o - Temperature.Thermal)
      (heated ?o - Temperature.Thermal))

   (:durative-action monitor
        :parameters (?o - Temperature.Thermal)
        :duration (= ?duration (planning-horizon))
        :condition (and
          (at start (can-start-monitoring))
          (at start (not (monitoring ?o)))
          (over all (>= (Temperature.current-temp ?o)
             (minimum-temp ?o)))
          (over all (<= (Temperature.current-temp ?o)
             (maximum-temp ?o)))
          (at end (not (Temperature.heating ?o))))
        :effect (and
          (at start (monitoring ?o))
          (at end (not (monitoring ?o)))
          (at end (monitored ?o))
          (decrease (Temperature.current-temp ?o)
                    (* #t (Temperature.cooling-rate ?o))))))

   (:durative-action heat
       :parameters (?o - Temperature.Thermal)
       :duration (>= ?duration 0)
       :condition (and
          (at start (monitoring ?o))
          (over all (monitoring ?o))
          (at end (monitoring ?o))
          (at start (not (Temperature.heating ?o)))
          (at start (<= (Temperature.current-temp ?o)
                        (thermostat-on-temp ?o)))
          (over all (<= (Temperature.current-temp ?o)
                        (maximum-temp ?o)))
          (at end (>= (Temperature.current-temp ?o)
                      (thermostat-minoff-temp ?o))))
       :effect (and
          (at start (Temperature.heating ?o))
          (increase (Temperature.current-temp ?o)
                    (* #t (heating-rate ?o)))
          (at end (not (Temperature.heating ?o)))
          (at end (not (Temperature.heating ?o))))))
```

Listing C.5: PDDLx Definition for the Thermostat Domain.

```
(define (problem thermostat-p7)
  (:domain thermostat)
  (:requirements
     :strips
     :typing
     :fluents
     :durative-actions
```

```
    :timed-initial-literals
    :timed-initial-fluents
    :negative-preconditions
    :duration-inequalities
    :class-modules)

  (:objects o1 - Temperature.Thermal)

  (:init
    (= (planning-horizon) 90)
    (= (Temperature.ambient-temp) -6)
    (= (Temperature.current-temp o1) 15)
    (= (Temperature.cooling-constant o1) 0.054)

    (= (minimum-temp o1) 10)
    (= (maximum-temp o1) 25)
    (= (heating-rate o1) 0.5)
    (= (thermostat-on-temp o1) 12)
    (= (thermostat-minoff-temp o1) 13)

    (at 0 (can-start-monitoring))
    (at 0.001 (not(can-start-monitoring)))

    (at 15 (= (Temperature.ambient-temp) -3))
    (at 30 (= (Temperature.ambient-temp) -1))
    (at 45 (= (Temperature.ambient-temp) 1))
    (at 60 (= (Temperature.ambient-temp) 2)))

 (:goal (and
   (monitored o1)
   (>= (Temperature.current-temp o1) (minimum-temp o1))
   (<= (Temperature.current-temp o1) (maximum-temp o1))))
)
```

Listing C.6: PDDLx Problem Instance for the Thermostat Domain.

C.3 Non-linear Planetary Rover

This domain is a variant of the Planetary Rover domain described in Section 4.10.3, with the main difference being that battery charging is non-linear, following a more realistic curve with diminishing returns. The rover needs to travel to waypoints and perform experiments on specific objectives at these waypoints, collecting data in the process. Navigating to waypoints and performing experiments depends on solar power available, which is predicted and modelled as a piecewise linear function with its gradient, `current-solar-power-dt`, updated with timed initial fluents. The planetary orbiter is scheduled to pass over waypoints at specific time intervals, which could coincide with periods when solar power is not available. The rover must be at the specific rendezvous to transmit data collected from experiments. It is also equipped with a battery, which must be charged during periods of solar power surplus, so that it can then be used to transmit the data.

The non-linear charging behaviour is modelled through an external class module, `Kcl.Planning.APS.Storage`, shown in Listing C.7, and semantically attached to the domain within the continuous effect of the `charge` action. This module introduces the types `Battery` and `Profile`, for domains where the battery can be charged with different charging profiles (such as normal or fast) with different wattage. In this case only one profile is used. The Non-Linear Cost and Storage Aggregator described in Appendix D.3 also makes use of this class module, and also support different battery charging profiles.

```
(define (module Kcl.Planning.APS.Storage)
  (:requirements :typing :fluents)
  (:types Battery Profile)
  (:predicates
    (mutable (available ?b - Battery))
        (init (charge-profile ?b - Battery ?p - Profile))
        (init (discharge-profile ?b - Battery ?p - Profile)))
  (:functions
    (init (full-charge-time ?b - Battery))
    (init (charge-power ?b - Battery ?p - Profile))
    (init (discharge-power ?b - Battery ?p - Profile))
    (init (discharge-rate ?b - Battery ?p - Profile))
    (mutable (state-of-charge ?b - Battery)))
  (:continuous-functions
    (charge-rate ?b - Battery ?p - Profile)))
```

Listing C.7: PDDLx Definition for Non-Linear Electricity Storage class module.

```
(define (domain nonlin-planetary-rover)
   (:requirements
      :strips
      :typing
      :fluents
      :durative-actions
      :timed-initial-literals
      :timed-initial-fluents
      :negative-preconditions
      :duration-inequalities
      :class-modules)
   (:classes Storage - Kcl.Planning.APS.Storage)
   (:types rover waypoint objective)
   (:predicates
      (can-start-measuring-power)
      (measuring-power ?r - rover)
      (measured-power ?r - rover)
      (operational ?r - rover)
      (rover-at ?r - rover ?w - waypoint)
      (can-travel ?from ?to - waypoint)
      (running-experiment ?o - objective)
      (objective-at ?o - objective ?w - waypoint)
      (complete ?o - objective)
      (travelling ?r - rover)
      (battery-profile ?b - Storage.Battery
                       ?p - Storage.Profile)
```

```
    (rover-battery ?r - rover ?b - Storage.Battery)
    (charging ?b - Storage.Battery)
    (can-transmit-from ?w - waypoint)
    (transmitted-data ?o - objective)
    (uplink-established ?r - rover)
    (transmitting ?r - rover))

(:functions
   (current-power ?r - rover)
   (power-needed-to-operate ?r - rover)
   (current-battery-power ?r - rover)
   (current-solar-power-dt ?r - rover)
   (power-needed-for-travel ?r - rover)
   (travel-time ?from ?to - waypoint)
   (experiment-duration ?r - rover ?o - objective)
   (power-needed-for-experiment ?r - rover ?o - objective)
   (battery-capacity ?b - Storage.Battery)
   (payload-transmission-duration ?o - objective)
   (power-needed-for-transmission ?r - rover)
   (establish-uplink-energy ?r - rover)
   (teardown-uplink-energy ?r - rover))

(:durative-action measure-power
    :parameters (?r - rover)
    :duration (>= ?duration 0)
    :condition (and (at start (not (measuring-power ?r)))
                    (at start (can-start-measuring-power)))
    :effect (and (at start (measuring-power ?r))
                 (at start (measured-power ?r))
                 (increase (current-power ?r)
                    (* #t (current-solar-power-dt ?r)))
                 (at end (not (measuring-power ?r)))))

(:durative-action operate
   :parameters (?r - rover)
   :duration (>= ?duration 0)
   :condition (and
    (at start (not(operational ?r)))
    (at start (>= (current-power ?r)
                  (power-needed-to-operate ?r)))
    (at start (measuring-power ?r))
    (over all (measuring-power ?r))
    (at end (measuring-power ?r))
    (over all (>= (current-power ?r) 0)))
  :effect (and
    (at start (decrease (current-power ?r)
                        (power-needed-to-operate ?r)))
    (at start (operational ?r))
    (at end (increase (current-power ?r)
                      (power-needed-to-operate ?r)))
    (at end (not (operational ?r)))))
```

```
(:durative-action navigate
  :parameters (?r - rover ?from ?to - waypoint)
  :duration (= ?duration (travel-time ?from ?to))
  :condition (and
    (at start (rover-at ?r ?from))
    (at start (can-travel ?from ?to))
    (at start (>= (current-power ?r)
                  (power-needed-for-travel ?r)))
    (at start (measuring-power ?r))
    (at start (not (travelling ?r)))
    (over all (measuring-power ?r))
    (at start (operational ?r))
    (over all (operational ?r))
    (over all (>= (current-power ?r) 0)))
  :effect (and
   (at start (decrease (current-power ?r)
                       (power-needed-for-travel ?r)))
   (at start (travelling ?r))
   (at start (not (rover-at ?r ?from)))
   (at end (rover-at ?r ?to))
   (at end (not (travelling ?r)))
   (at end (increase (current-power ?r)
                     (power-needed-for-travel ?r)))))

(:durative-action experiment
  :parameters (?r - rover ?w - waypoint ?o - objective)
  :duration (= ?duration (experiment-duration ?r ?o))
  :condition (and
    (at start (not (complete ?o)))
    (at start (not (running-experiment ?o)))
    (at start (objective-at ?o ?w))
    (at start (rover-at ?r ?w))
    (at start (operational ?r))
    (at start (>= (current-power ?r)
                  (power-needed-for-experiment ?r ?o)))
    (at start (measuring-power ?r))
    (over all (measuring-power ?r))
    (at end (measuring-power ?r))
    (over all (operational ?r))
    (over all (not (travelling ?r)))
    (over all (>= (current-power ?r) 0)))
  :effect (and
    (at start (running-experiment ?o))
    (at start (decrease (current-power ?r)
                (power-needed-for-experiment ?r ?o)))
    (at end (increase (current-power ?r)
              (power-needed-for-experiment ?r ?o)))
    (at end (not (running-experiment ?o)))
    (at end (complete ?o))))

(:durative-action establish-uplink
  :parameters (?r - rover
```

```
                ?b - Storage.Battery
                ?w - waypoint)
  :duration (>= ?duration 0)
  :condition (and
    (at start (rover-battery ?r ?b))
    (at start (rover-at ?r ?w))
    (at start (can-transmit-from ?w))
    (at start (not (charging ?b)))
    (at start (not (uplink-established ?r)))
    (at start (measuring-power ?r))
    (over all (measuring-power ?r))
    (at end (measuring-power ?r))
    (over all (not (travelling ?r)))
    (over all (>= (Storage.state-of-charge ?b) 0)))
  :effect (and
    (at start (uplink-established ?r))
    (decrease (Storage.state-of-charge ?b)
              (* #t (/ (power-needed-for-transmission ?r)
                       (battery-capacity ?b))))
    (at end (not (uplink-established ?r)))))

(:durative-action transmit-experiment-data
   :parameters (?r - rover ?o - objective)
   :duration (= ?duration
                (payload-transmission-duration ?o))
   :condition (and
     (at start (not (transmitting ?r)))
     (at start (complete ?o))
     (at start (not (transmitted-data ?o)))
     (at start (measuring-power ?r))
     (over all (measuring-power ?r))
     (at end (measuring-power ?r))
     (at start (uplink-established ?r))
     (over all (uplink-established ?r)))
   :effect (and
     (at start (transmitting ?r))
     (at end (not (transmitting ?r)))
     (at end (transmitted-data ?o))))

(:durative-action charge
  :parameters (?r - rover
               ?b - Storage.Battery
               ?p - Storage.Profile)
  :duration (>= ?duration 0)
  :condition (and
    (at start (rover-battery ?r ?b))
    (at start (battery-profile ?b ?p))
    (at start (>= (current-power ?r)
                  (Storage.charge-power ?b ?p)))
    (at start (<= (Storage.state-of-charge ?b) 99))
    (at start (not (charging ?b)))
    (at start (measuring-power ?r))
```

```
      (over all (measuring-power ?r))
      (at end (measuring-power ?r))
      (over all (>= (current-power ?r) 0))
      (over all (<= (Storage.state-of-charge ?b) 100)))
    :effect (and
      (at start (charging ?b))
      (at start (increase (current-power ?r)
                          (Storage.charge-power ?b ?p)))
      (increase (Storage.state-of-charge ?b)
                (* #t (Storage.charge-rate ?b ?p)))
      (at end (decrease (current-power ?r)
                        (Storage.charge-power ?b ?p)))
      (at end (not (charging ?b)))))
)
```

Listing C.8: PDDLx Definition for the Non-Linear Planetary Rover Domain.

```
(define (problem nonlin-planetary-rover-p6)
  (:domain nonlin-planetary-rover)
  (:requirements
      :strips
      :typing
      :fluents
      :durative-actions
      :timed-initial-literals
      :timed-initial-fluents
      :negative-preconditions
      :duration-inequalities
      :class-modules)

  (:objects
      rover1 - rover
      battery1 - Storage.Battery
      profile - Storage.Profile
      w0 w1 w2 w3 w4 w5 - waypoint
      ob1 ob2 ob3 ob4 ob5 ob6 - objective)

 (:init
    (can-travel w0 w1)
    (= (travel-time w0 w1) 1)
    (can-travel w1 w0)
    (= (travel-time w1 w0) 1)
    (can-travel w1 w2)
    (= (travel-time w1 w2) 1)
    (can-travel w2 w1)
    (= (travel-time w2 w1) 1)
    (can-travel w0 w2)
    (= (travel-time w0 w2) 1.5)
    (can-travel w2 w0)
    (= (travel-time w2 w0) 1.5)
    (can-travel w1 w3)
    (= (travel-time w1 w3) 1)
    (can-travel w3 w1)
```

```
(= (travel-time w3 w1) 1)
(can-travel w2 w3)
(= (travel-time w2 w3) 1)
(can-travel w3 w2)
(= (travel-time w3 w2) 1)
(can-travel w4 w2)
(= (travel-time w4 w2) 2)
(can-travel w2 w4)
(= (travel-time w2 w4) 1)
(can-travel w4 w3)
(= (travel-time w4 w3) 1)
(can-travel w3 w4)
(= (travel-time w3 w4) 1)
(can-travel w3 w5)
(= (travel-time w3 w5) 0.5)
(can-travel w5 w3)
(= (travel-time w5 w3) 0.5)
(can-travel w4 w5)
(= (travel-time w4 w5) 0.5)
(can-travel w5 w4)
(= (travel-time w5 w4) 0.5)
(can-travel w2 w5)
(= (travel-time w2 w5) 0.5)
(can-travel w5 w2)
(= (travel-time w5 w2) 0.5)

(can-start-measuring-power)
(at 0.001 (not(can-start-measuring-power)))
(at 18 (can-transmit-from w2))
(at 21 (not (can-transmit-from w2)))
(at 20 (can-transmit-from w5))
(at 23 (not (can-transmit-from w5)))
(at 23 (can-transmit-from w3))
(at 25 (not (can-transmit-from w3)))

(at 6 (= (current-solar-power-dt rover1) 5))
(at 10 (= (current-solar-power-dt rover1) 3))
(at 11 (= (current-solar-power-dt rover1) 0))
(at 13 (= (current-solar-power-dt rover1) -3))
(at 14 (= (current-solar-power-dt rover1) -5))
(at 18 (= (current-solar-power-dt rover1) 0))

(battery-profile battery1 profile)
(rover-battery rover1 battery1)
(= (power-needed-to-operate rover1) 1)
(= (power-needed-for-travel rover1) 0.2)
(= (current-power rover1) 0)
(= (current-solar-power-dt rover1) 0)
(= (battery-capacity battery1) 20)
(= (Storage.charge-power battery1 profile) 2)
(= (Storage.full-charge-time battery1 profile) 2)
(= (Storage.state-of-charge battery1) 0)
```

```
    (= (power-needed-for-transmission rover1) 1)
    (rover-at rover1 w0)

    (objective-at ob1 w1)
    (= (payload-transmission-duration ob1) 0.1)
    (= (experiment-duration rover1 ob1) 2)
    (= (power-needed-for-experiment rover1 ob1) 2)

    (objective-at ob2 w2)
    (= (payload-transmission-duration ob2) 0.1)
    (= (experiment-duration rover1 ob2) 2)
    (= (power-needed-for-experiment rover1 ob2) 2)

    (objective-at ob3 w2)
    (= (payload-transmission-duration ob3) 0.1)
    (= (experiment-duration rover1 ob3) 1)
    (= (power-needed-for-experiment rover1 ob3) 3)

    (objective-at ob4 w3)
    (= (payload-transmission-duration ob4) 0.1)
    (= (experiment-duration rover1 ob4) 0.5)
    (= (power-needed-for-experiment rover1 ob4) 1)

    (objective-at ob5 w1)
    (= (payload-transmission-duration ob5) 0.4)
    (= (experiment-duration rover1 ob5) 1)
    (= (power-needed-for-experiment rover1 ob5) 4)

    (objective-at ob6 w5)
    (= (payload-transmission-duration ob6) 0.2)
    (= (experiment-duration rover1 ob6) 0.3)
    (= (power-needed-for-experiment rover1 ob6) 3))

 (:goal (and
     (transmitted-data ob1)
     (transmitted-data ob2)
     (transmitted-data ob3)
     (transmitted-data ob4)
     (transmitted-data ob5)
     (transmitted-data ob6)))
)
```

Listing C.9: PDDLx Problem Instance for the Non-Linear Planetary Rover Domain.

D. The Aggregator Domain

This appendix contains full definitions of the respective domain models and problem instances for the case study in Chapter 7. For each domain model a sample problem instance is provided to demonstrate a full working example.

D.1 Linear Aggregator

In this model, the `unit-price` is modelled as a step function, changing every hour. The `inflexible-load` and `flexible-cost` are modelled independently as piecewise linear functions. The goal involves both propositional conditions (performing an activity) and numeric goals (charging a battery to the required level).

```
(define (domain linear-aggregator)
  (:requirements
     :strips
     :typing
     :fluents
     :durative-actions
     :negative-preconditions
     :duration-inequalities
     :timed-initial-literals
     :timed-initial-fluents)

  (:types activity profile battery)

  (:predicates
    (can-start-metering)
    (metered)
    (metering)

    (activity-profile ?a - activity ?p - profile)
    (can-perform ?a - activity)
    (performed ?a - activity)

    (charge-profile ?b - battery ?p - profile)
    (discharge-profile ?b - battery ?p - profile)
    (available ?b - battery)
    (in-use ?b - battery))


  (:functions
```

```
  (inflexible-load)
  (inflexible-load-dt) ;gradient of inflexible load
  (flexible-load)
  (max-load) ;maximum load
  (min-load) ;minimum load
  (power-needed ?a - activity ?p - profile)
  (duration-needed ?a - activity ?p - profile)
  (state-of-charge ?b - battery)
  (charge-power ?b - battery ?p - profile)
  (discharge-power ?b - battery ?p - profile)
  (charge-rate ?b - battery ?p - profile)
  (discharge-rate ?b - battery ?p - profile)
  (unit-price)
  (flexible-cost))

;envelope action to perform continuous metering
(:durative-action meter
  :parameters()
  :duration (>= ?duration 0)
  :condition (and
    (at start (can-start-metering))
    (at start (not (metering)))
    (at start (not (metered)))
    (over all (<= (min-load) (+ (inflexible-load)
                                (flexible-load))))
    (over all (>= (max-load) (+ (inflexible-load)
                                (flexible-load)))))
  :effect (and
    (at start (metering))
    (at start (metered))
    (at end (not (metering)))
    (increase (inflexible-load)
              (* #t (inflexible-load-dt)))
    (increase (flexible-cost)
              (* #t (* (unit-price) (flexible-load))))))

;action that performs a flexible activity
(:durative-action perform
  :parameters (?a - activity ?p - profile)
  :duration (= ?duration (duration-needed ?a ?p))
  :condition (and
    (at start (not(performed ?a)))
    (at start (activity-profile ?a ?p))
    (over all (can-perform ?a))
    (over all (metering)))
  :effect (and
    (at start (increase (flexible-load)
                        (power-needed ?a ?p)))
    (at end (decrease (flexible-load)
                      (power-needed ?a ?p)))
    (at end (performed ?a))))
```

```
  ;action that charges a battery
  (:durative-action charge
    :parameters (?b - battery ?p - profile)
    :duration (>= ?duration 0)
    :condition (and
      (at start (not (in-use ?b)))
      (at start (charge-profile ?b ?p))
      (over all (<= (state-of-charge ?b) 100))
      (over all (available ?b))
      (over all (metering)))
    :effect (and
      (at start (in-use ?b))
      (at start (increase (flexible-load)
                          (charge-power ?b ?p)))
      (at end (decrease (flexible-load)
                        (charge-power ?b ?p)))
      (at end (not (in-use ?b)))
      (increase (state-of-charge ?b)
                (* #t (charge-rate ?b ?p)))))

  ;action that discharges a battery
  (:durative-action discharge
    :parameters (?b - battery ?p - profile)
    :duration (>= ?duration 0)
    :condition (and
      (at start (not (in-use ?b)))
      (at start (discharge-profile ?b ?p))
      (over all (>= (state-of-charge ?b) 0))
      (over all (available ?b))
      (over all (metering)))
    :effect (and
      (at start (in-use ?b))
      (at start (decrease (flexible-load)
                          (discharge-power ?b ?p)))
      (at end (increase (flexible-load)
                        (discharge-power ?b ?p)))
      (at end (not (in-use ?b)))
      (decrease (state-of-charge ?b)
                (* #t (discharge-rate ?b ?p)))))
)
```

Listing D.1: PDDL Domain File for the Linear Aggregator.

```
(define (problem linear-aggregator-p3)
  (:domain linear-aggregator)

  (:requirements
        :strips
        :typing
        :fluents
        :durative-actions
        :timed-initial-literals
        :timed-initial-fluents
        :negative-preconditions
        :duration-inequalities)

  (:objects dishwasher1-h1 dishwasher1-h2 dishwasher1-h3 -
     activity
            normal - profile
            b1 - battery)

  (:init
    (= (inflexible-load) 400)
    (= (inflexible-load-dt) -6.667)
    (= (max-load) 10000)
    (= (min-load) 100)

    (= (flexible-load) 0)
    (= (flexible-cost) 0)
    (= (unit-price) 1.0)

    (= (state-of-charge b1) 30)
    (available b1)

    (charge-profile b1 normal)
    (discharge-profile b1 normal)
    (= (charge-power b1 normal) 2.0)
    (= (discharge-power b1 normal) 1.9)
    (= (charge-rate b1 normal) 0.5)
    (= (discharge-rate b1 normal) 0.7)

    ;clip for timer envelope action
    (at 0 (can-start-metering))
    (at 0.001 (not (can-start-metering)))

    (at 15 (= (inflexible-load-dt) 1.333))
    (at 30 (= (inflexible-load-dt) 0))

    (activity-profile dishwasher1-h1 normal)
    (= (power-needed dishwasher1-h1 normal) 3.3)
    (= (duration-needed dishwasher1-h1 normal) 420)

    (at 120 (can-perform dishwasher1-h1))
    (at 600 (not(can-perform dishwasher1-h1)))
```

```
    (activity-profile dishwasher1-h2 normal)
    (= (power-needed dishwasher1-h2 normal) 3.3)
    (= (duration-needed dishwasher1-h2 normal) 420)

    (at 200 (can-perform dishwasher1-h2))
    (at 1000 (not(can-perform dishwasher1-h2)))

    (activity-profile dishwasher1-h3 normal)
    (= (power-needed dishwasher1-h3 normal) 2.3)
    (= (duration-needed dishwasher1-h3 normal) 420)

    (activity-profile dishwasher1-h3 fast)
    (= (power-needed dishwasher1-h3 fast) 4.3)
    (= (duration-needed dishwasher1-h3 fast) 220)

    (at 600 (can-perform dishwasher1-h3))
    (at 900 (not(can-perform dishwasher1-h3)))

    (at 400 (= (unit-price) 0.7)))

 (:goal (and
   (performed dishwasher1-h1)
   (performed dishwasher1-h2)
   (performed dishwasher1-h3)

   (>= (state-of-charge b1) 90)))

 (:metric minimize (flexible-cost))
)
```

Listing D.2: PDDL Problem File for the Linear Aggregator.

D.2 Non-Linear Cost Aggregator

In this model, `total-cost` takes into account both `inflexible-load` and `flexible-load`. It encapsulates the non-linear rate of change of this cost with the `Kcl.Planning.APS.Aggregator` external *class module*, defined in Listing 7.15.

```
(define (domain nonlincost-aggregator)
  (:requirements
     :strips
     :typing
     :fluents
     :durative-actions
     :negative-preconditions
     :duration-inequalities
     :timed-initial-literals
     :timed-initial-fluents
     :class-modules)

  (:classes Aggregator - Kcl.Planning.APS.Aggregator)
  (:types activity profile battery)
```

```
(:predicates
  (can-start-metering)
  (metered)
  (metering)
  (activity-profile ?a - activity ?p - profile)
  (can-perform ?a - activity)
  (performed ?a - activity)
  (charge-profile ?b - battery ?p - profile)
  (discharge-profile ?b - battery ?p - profile)
  (available ?b - battery)
  (in-use ?b - battery))

(:functions
  (max-load) ;maximum load
  (min-load) ;minimum load
  (power-needed ?a - activity ?p - profile)
  (duration-needed ?a - activity ?p - profile)
  (state-of-charge ?b - battery)
  (charge-power ?b - battery ?p - profile)
  (discharge-power ?b - battery ?p - profile)
  (charge-rate ?b - battery ?p - profile)
  (discharge-rate ?b - battery ?p - profile))

;envelope action to perform continuous metering
(:durative-action meter
  :parameters()
  :duration (>= ?duration 0)
  :condition (and
    (at start (can-start-metering))
    (at start (not (metering)))
    (at start (not (metered)))
    (over all (<= (min-load)
                  (+ (Aggregator.inflexible-load)
                     (Aggregator.flexible-load))))
    (over all (>= (max-load)
                  (+ (Aggregator.inflexible-load)
                     (Aggregator.flexible-load)))))
  :effect (and
    (at start (metering))
    (at start (metered))
    (at end (not (metering)))
    (increase (Aggregator.inflexible-load)
              (* #t (Aggregator.inflexible-load-dt)))
    (increase (Aggregator.total-cost)
              (* #t Aggregator.total-cost-dt))))

;action that performs a flexible activity
(:durative-action perform
  :parameters (?a - activity ?p - profile)
  :duration (= ?duration (duration-needed ?a ?p))
  :condition (and
```

```
      (at start (not(performed ?a)))
      (at start (activity-profile ?a ?p))
      (over all (can-perform ?a))
      (over all (metering)))
    :effect (and
      (at start (increase (Aggregator.flexible-load)
                          (power-needed ?a ?p)))
      (at end (decrease (Aggregator.flexible-load)
                        (power-needed ?a ?p)))
      (at end (performed ?a))))

  ;action that charges a battery
  (:durative-action charge
    :parameters (?b - battery ?p - profile)
    :duration (>= ?duration 0)
    :condition (and
      (at start (not (in-use ?b)))
      (at start (charge-profile ?b ?p))
      (over all (<= (state-of-charge ?b) 100))
      (over all (available ?b))
      (over all (metering)))
    :effect (and
      (at start (in-use ?b))
      (at start (increase (Aggregator.flexible-load)
                          (charge-power ?b ?p)))
      (at end (decrease (Aggregator.flexible-load)
                        (charge-power ?b ?p)))
      (at end (not (in-use ?b)))
      (increase (state-of-charge ?b)
                (* #t (charge-rate ?b ?p)))))

  ;action that discharges a battery
  (:durative-action discharge
    :parameters (?b - battery ?p - profile)
    :duration (>= ?duration 0)
    :condition (and
      (at start (not (in-use ?b)))
      (at start (discharge-profile ?b ?p))
      (over all (>= (state-of-charge ?b) 0))
      (over all (available ?b))
      (over all (metering)))
    :effect (and
      (at start (in-use ?b))
      (at start (decrease (Aggregator.flexible-load)
                          (discharge-power ?b ?p)))
      (at end (increase (Aggregator.flexible-load)
                        (discharge-power ?b ?p)))
      (at end (not (in-use ?b)))
      (decrease (state-of-charge ?b)
                (* #t (discharge-rate ?b ?p)))))
)
```

Listing D.3: PDDLx Domain File for the Non-Linear Cost Aggregator.

```
(define (problem nonlincost-aggregator-p3)
  (:domain nonlincost-aggregator)

  (:requirements
        :strips
        :typing
        :fluents
        :durative-actions
        :negative-preconditions
        :duration-inequalities
        :timed-initial-literals
        :timed-initial-fluents
        :class-modules)

  (:objects dishwasher1-h1 dishwasher1-h2 - activity
            dishwasher1-h3 - activity
            normal - profile
            b1 - battery)

  (:init
    (= (Aggregator.inflexible-load) 400)
    (= (Aggregator.inflexible-load-dt) -6.667)
    (= (max-load) 10000)
    (= (min-load) 100)

    (= (Aggregator.flexible-load) 0)
    (= (Aggregator.total-cost) 0)
    (= (Aggregator.unit-price) 1.0)

    (= (state-of-charge b1) 30)
    (available b1)

    (charge-profile b1 normal)
    (discharge-profile b1 normal)
    (= (charge-power b1 normal) 2.0)
    (= (discharge-power b1 normal) 1.9)
    (= (charge-rate b1 normal) 0.5)
    (= (discharge-rate b1 normal) 0.7)

    ;clip for timer envelope action
    (at 0 (can-start-metering))
    (at 0.001 (not (can-start-metering)))

    (at 15 (= (Aggregator.inflexible-load-dt) 1.333))
    (at 30 (= (Aggregator.inflexible-load-dt) 0))

    (activity-profile dishwasher1-h1 normal)
    (= (power-needed dishwasher1-h1 normal) 3.3)
    (= (duration-needed dishwasher1-h1 normal) 420)

    (at 120 (can-perform dishwasher1-h1))
    (at 600 (not(can-perform dishwasher1-h1)))
```

```
    (activity-profile dishwasher1-h2 normal)
    (= (power-needed dishwasher1-h2 normal) 3.3)
    (= (duration-needed dishwasher1-h2 normal) 420)

    (at 200 (can-perform dishwasher1-h2))
    (at 1000 (not(can-perform dishwasher1-h2)))

    (activity-profile dishwasher1-h3 normal)
    (= (power-needed dishwasher1-h3 normal) 2.3)
    (= (duration-needed dishwasher1-h3 normal) 420)

    (activity-profile dishwasher1-h3 fast)
    (= (power-needed dishwasher1-h3 fast) 4.3)
    (= (duration-needed dishwasher1-h3 fast) 220)

    (at 600 (can-perform dishwasher1-h3))
    (at 900 (not(can-perform dishwasher1-h3)))

    (at 400 (= (Aggregator.unit-price) 0.7)))

  (:goal (and
    (performed dishwasher1-h1)
    (performed dishwasher1-h2)
    (performed dishwasher1-h3)
    (>= (state-of-charge b1) 90)))

  (:metric minimize (Aggregator.total-cost))
)
```

Listing D.4: PDDLx Problem File for the Non-Linear Cost Aggregator.

D.3 Non-Linear Cost and Storage Aggregator

In this model, apart from having a non-linear `total-cost`, the storage facilities also charge non-linearly. This is encapsulated through the `Kcl.Planning.APS.Storage` external *class module*, defined in Listing 7.16.

```
(define (domain nonlinear-aggregator)
  (:requirements
     :strips
     :typing
     :fluents
     :durative-actions
     :negative-preconditions
     :duration-inequalities
     :timed-initial-literals
     :timed-initial-fluents
     :class-modules)

  (:classes Aggregator - Kcl.Planning.APS.Aggregator
            Storage - Kcl.Planning.APS.Storage)
```

```
(:types activity
        profile - Storage.Profile)

(:predicates
  (can-start-metering)
  (metered)
  (metering)
  (in-use ?b - Storage.Battery)
  (activity-profile ?a - activity ?p - profile)
  (can-perform ?a - activity)
  (performed ?a - activity))

(:functions
  (max-load) ;maximum load
  (min-load) ;minimum load
  (power-needed ?a - activity ?p - profile)
  (duration-needed ?a - activity ?p - profile))

;envelope action to perform continuous metering
(:durative-action meter
  :parameters()
  :duration (>= ?duration 0)
  :condition (and
    (at start (can-start-metering))
    (at start (not (metering)))
    (at start (not (metered)))
    (over all (<= (min-load)
                  (+ (Aggregator.inflexible-load)
                     (Aggregator.flexible-load))))
    (over all (>= (max-load)
                  (+ (Aggregator.inflexible-load)
                     (Aggregator.flexible-load)))))
  :effect (and
    (at start (metering))
    (at start (metered))
    (at end (not (metering)))
    (increase (Aggregator.inflexible-load)
              (* #t (Aggregator.inflexible-load-dt)))
    (increase (Aggregator.total-cost)
              (* #t Aggregator.total-cost-dt))))

;action that performs a flexible activity
(:durative-action perform
  :parameters (?a - activity ?p - profile)
  :duration (= ?duration (duration-needed ?a ?p))
  :condition (and
    (at start (not(performed ?a)))
    (at start (activity-profile ?a ?p))
    (over all (can-perform ?a))
    (over all (metering)))
  :effect (and
```

```
      (at start (increase (Aggregator.flexible-load)
                          (power-needed ?a ?p)))
      (at end (decrease (Aggregator.flexible-load)
                        (power-needed ?a ?p)))
      (at end (performed ?a))))

  ;action that charges a battery
  (:durative-action charge
    :parameters (?b - Storage.Battery ?p - Storage.Profile)
    :duration (>= ?duration 0)
    :condition (and
      (at start (not (in-use ?b)))
      (at start (Storage.charge-profile ?b ?p))
      (over all (<= (Storage.state-of-charge ?b) 100))
      (over all (Storage.available ?b))
      (over all (metering)))
    :effect (and
      (at start (in-use ?b))
      (at start (increase (Aggregator.flexible-load)
                          (Storage.charge-power ?b ?p)))
      (at end (decrease (Aggregator.flexible-load)
                        (Storage.charge-power ?b ?p)))
      (at end (not (in-use ?b)))
      (increase (Storage.state-of-charge ?b)
                (* #t (Storage.charge-rate ?b ?p))))))

  ;action that discharges a battery
  (:durative-action discharge
    :parameters (?b - Storage.Battery ?p - Storage.Profile)
    :duration (>= ?duration 0)
    :condition (and
      (at start (not (in-use ?b)))
      (at start (Storage.discharge-profile ?b ?p))
      (over all (>= (Storage.state-of-charge ?b) 0))
      (over all (Storage.available ?b))
      (over all (metering)))
    :effect (and
      (at start (in-use ?b))
      (at start (decrease (Aggregator.flexible-load)
                          (Storage.discharge-power ?b ?p)))
      (at end (increase (Aggregator.flexible-load)
                        (Storage.discharge-power ?b ?p)))
      (at end (not (in-use ?b)))
      (decrease (Storage.state-of-charge ?b)
                (* #t (Storage.discharge-rate ?b ?p))))))
)
```

Listing D.5: PDDLx Domain File for the Non-Linear Cost and Storage Aggregator.

```
(define (problem nonlinear-aggregator-p3)
  (:domain nonlinear-aggregator)

  (:requirements
        :strips
        :typing
        :fluents
        :durative-actions
        :negative-preconditions
        :duration-inequalities
        :timed-initial-literals
        :timed-initial-fluents
        :class-modules)

  (:objects dishwasher1-h1 dishwasher1-h2 - activity
            dishwasher1-h3 - activity
            normal - profile
            b1 - Storage.Battery)

  (:init
    (= (Aggregator.inflexible-load) 400)
    (= (Aggregator.inflexible-load-dt) -6.667)
    (= (max-load) 10000)
    (= (min-load) 100)

    (= (Aggregator.flexible-load) 0)
    (= (Aggregator.total-cost) 0)
    (= (Aggregator.unit-price) 1.0)

    (= (state-of-charge b1) 30)
    (available b1)

    (Storage.charge-profile b1 normal)
    (Storage.discharge-profile b1 normal)
    (= (Storage.charge-power b1 normal) 2.0)
    (= (Storage.discharge-power b1 normal) 1.9)
    (= (Storage.discharge-rate b1 normal) 0.7)

    ;clip for timer envelope action
    (at 0 (can-start-metering))
    (at 0.001 (not (can-start-metering)))

    (at 15 (= (Aggregator.inflexible-load-dt) 1.333))
    (at 30 (= (Aggregator.inflexible-load-dt) 0))

    (activity-profile dishwasher1-h1 normal)
    (= (power-needed dishwasher1-h1 normal) 3.3)
    (= (duration-needed dishwasher1-h1 normal) 420)

    (at 120 (can-perform dishwasher1-h1))
    (at 600 (not(can-perform dishwasher1-h1)))
```

```
    (activity-profile dishwasher1-h2 normal)
    (= (power-needed dishwasher1-h2 normal) 3.3)
    (= (duration-needed dishwasher1-h2 normal) 420)

    (at 200 (can-perform dishwasher1-h2))
    (at 1000 (not(can-perform dishwasher1-h2)))

    (activity-profile dishwasher1-h3 normal)
    (= (power-needed dishwasher1-h3 normal) 2.3)
    (= (duration-needed dishwasher1-h3 normal) 420)

    (activity-profile dishwasher1-h3 fast)
    (= (power-needed dishwasher1-h3 fast) 4.3)
    (= (duration-needed dishwasher1-h3 fast) 220)

    (at 600 (can-perform dishwasher1-h3))
    (at 900 (not(can-perform dishwasher1-h3)))

    (at 400 (= (Aggregator.unit-price) 0.7)))

 (:goal (and
   (performed dishwasher1-h1)
   (performed dishwasher1-h2)
   (performed dishwasher1-h3)

   (>= (Storage.state-of-charge b1) 90)))

 (:metric minimize (Aggregator.total-cost))
)
```

Listing D.6: PDDLx Problem File for the Non-Linear Cost and Storage Aggregator.

Bibliography

Aker, Erdi, Volkan Patoglu, and Esra Erdem (2012). "Answer Set Programming for Reasoning with Semantic Knowledge in Collaborative Housekeeping Robotics". In: *Proceedings of the 10th IFAC Symposium on Robot Control*, pp. 77–83.

Allen, James F. (1983). "Maintaining Knowledge about Temporal Intervals". In: *Communications of the ACM* 26.11, pp. 832–843.

APX Group (2016). *UK Energy Market Results*. https://www.apxgroup.com/market-results/apx-power-uk/dashboard/.

Bajada, Josef, Maria Fox, and Derek Long (2013). "Load Modelling and Simulation of Household Electricity Consumption for the Evaluation of Demand-Side Management Strategies". In: *4th IEEE PES Innovative Smart Grid Technologies Europe (ISGT Europe). IEEE*.

Bajada, Josef, Maria Fox, and Derek Long (2014a). "Challenges in Temporal Planning for Aggregate Load Management of Household Electricity Demand". In: *31st Workshop of the UK Planning & Scheduling Special Interest Group (PlanSIG)*.

Bajada, Josef, Maria Fox, and Derek Long (2014b). "Temporal Plan Quality Improvement and Repair using Local Search". In: *Proceedings of the 7th European Starting A.I. Researcher Symposium (STAIRS-2014)*. IOS Press, pp. 41–50.

Bajada, Josef, Maria Fox, and Derek Long (2015). "Temporal Planning with Semantic Attachment of Non-Linear Monotonic Continuous Behaviours". In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI-15)*, pp. 1523–1529.

Bajada, Josef, Maria Fox, and Derek Long (2016). "Temporal Planning with Constants in Context". In: *Proceedings of the 22nd European Conference on Artificial Intelligence (ECAI 2016)*.

Barker, Sean, Aditya Mishra, David Irwin, Prashant Shenoy, and Jeannie Albrecht (2012). "SmartCap: Flattening peak electricity demand in smart homes". In: *2012 IEEE International Conference on Pervasive Computing and Communications*. IEEE, pp. 67–75.

Benton, J., Amanda Coles, and Andrew Coles (2012). "Temporal Planning with Preferences and Time-Dependent Continuous Costs". In: *Proceedings of the 22nd International Conference on Automated Planning and Scheduling*, pp. 2–10.

Blum, Avrim L. and Merrick L. Furst (1997). “Fast Planning Through Planning Graph Analysis”. In: *Artificial Intelligence* 90.1-2, pp. 281–300.

Bogomolov, Sergiy, Daniele Magazzeni, Andreas Podelski, and Martin Wehrle (2014). “Planning as Model Checking in Hybrid Domains”. In: *Proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI-14)*, pp. 2228–2234.

Bonet, Blai and Héctor Geffner (2001). “Planning as heuristic search”. In: *Artificial Intelligence* 129, pp. 5–33.

Bonet, Blai, Gábor Loerincs, and Héctor Geffner (1997). “A Robust and Fast Action Selection Mechanism for Planning”. In: *Proceedings of the 14th National Conference of the American Association for Artificial Intelligence (AAAI-97)*. MIT Press, pp. 714–719.

Brooks, Alec, Ed Lu, Dan Reicher, Charles Spirakis, and Bill Weihl (2010). “Demand Dispatch”. In: *IEEE Power and Energy Magazine* 8.3, pp. 20–29.

Bryce, Daniel, Sicun Gao, David Musliner, and Robert Goldman (2015). “SMT-Based Nonlinear PDDL+ Planning”. In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI-15)*.

Busquet, Ana Rosselló, Georgios Kardaras, Villy Bæk Iversen, José Soler, and Lars Dittmann (2011). “Reducing Electricity Demand Peaks by Scheduling Home Appliances Usage”. In: *Risø International Energy Conference 2011*, pp. 156–163.

Bylander, Tom (1994). “The Computational Complexity of Propositional STRIPS Planning”. In: *Artificial Intelligence* 69.1-2, pp. 165–204.

Chan, H. L. and D. Sutanto (2000). “A new battery model for use with battery energy storage systems and electric vehicles power systems”. In: *2000 IEEE Power Engineering Society Winter Meeting*. Vol. 1. IEEE, pp. 470–475.

Coles, Amanda and Andrew Coles (2014). “PDDL+ Planning with Events and Linear Processes”. In: *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS-2014)*, pp. 74–82.

Coles, Amanda, Andrew Coles, Maria Fox, and Derek Long (2009a). “Extending the Use of Inference in Temporal Planning as Forwards Search”. In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-2009)*. Vol. 1.

Coles, Amanda, Andrew Coles, Maria Fox, and Derek Long (2009b). “Temporal Planning in Domains with Linear Processes”. In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*.

Coles, Amanda, Andrew Coles, Maria Fox, and Derek Long (2010). “Forward-Chaining Partial-Order Planning”. In: *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-2010)*, pp. 42–49.

Coles, Amanda, Andrew Coles, Maria Fox, and Derek Long (2012). "COLIN: Planning with Continuous Linear Numeric Change". In: *Journal of Artificial Intelligence Research* 44, pp. 1–96.

Coles, Andrew, Maria Fox, Derek Long, and Amanda Smith (2008). "Planning with Problems Requiring Temporal Coordination". In: *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pp. 892–897.

Coles, Andrew, Maria Fox, Keith Halsey, Derek Long, and Amanda Smith (2009c). "Managing Concurrency in Temporal Planning using Planner-Scheduler Interaction". In: *Artificial Intelligence* 173.1, pp. 1–44.

Cushing, William, Subbarao Kambhampati, Mausam, and Daniel S. Weld (2007). "When is Temporal Planning Really Temporal?" In: *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*.

Dechter, Rina, Itay Meiri, and Judea Pearl (1991). "Temporal constraint networks". In: *Artificial Intelligence* 49.1-3, pp. 61–95.

Della Penna, Giuseppe, Daniele Magazzeni, Fabio Mercorio, and Benedetto Intrigila (2009). "UPMurphi: A Tool for Universal Planning on PDDL+ Problems." In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-2009)*. AAAI, pp. 106–113.

Della Penna, Giuseppe, Daniele Magazzeni, and Fabio Mercorio (2012). "A Universal Planning System for Hybrid Domains". In: *Applied Intelligence* 36.4, pp. 932–959.

Dijkstra, Edsger Wybe (1959). "A Note on Two Problems in Connexion with Graphs". In: *Numerische Mathematik* 1.1, pp. 269–271.

Do, Minh B and Subbarao Kambhampati (2003a). "Sapa : A Multi-objective Metric Temporal Planner". In: *Journal of Artificial Intelligence Research* 20, pp. 155–194.

Do, Minh B and Subbarao Kambhampati (2003b). "Sapa: A Multi-objective Metric Temporal Planner". In: *Journal of Artificial Intelligence Research* 20, pp. 155–194.

Dornhege, Christian, Patrick Eyerich, Thomas Keller, Sebastian Trüg, Michael Brenner, and Bernhard Nebel (2009). "Semantic Attachments for Domain-Independent Planning Systems". In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-2009)*, pp. 114–121.

Edelkamp, Stefan (2001). "Planning with pattern databases". In: *Proceedings of the 6th European Conference on Planning (ECP-01)*, pp. 13–34.

Edelkamp, Stefan and Jörg Hoffmann (2004). *PDDL2.2: The Language for the Classical Part of the 4th International Planning Competition*. Tech. rep. 195, pp. 1–21.

Edelkamp, Stefan and Shahid Jabbar (2006). "Cost-Optimal External Planning". In: *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*. AAAI Press; MIT Press, pp. 821–826.

Edelkamp, Stefan and Shahid Jabbar (2008). “MIPS-XXL : Featuring External Shortest Path Search for Sequential Optimal Plans and External Branch-And-Bound for Optimal Net Benefit”. In: *The 6th International Planning Competition*.

Eiter, Thomas, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits (2006). “Effective Integration of Declarative Rules with External Evaluations for Semantic-Web Reasoning”. In: *The Semantic Web: Research and Applications. Proceedings of the 3rd European Semantic Web Conference (ESWC 2006)*. Springer Berlin Heidelberg, pp. 273–287.

Erdem, Esra, Kadir Haspalamutgil, Can Palaz, Volkan Patoglu, and Tansel Uras (2011). “Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation”. In: *2011 IEEE International Conference on Robotics and Automation*. IEEE, pp. 4575–4581.

Eyerich, Patrick, R Mattmüller, and G Röger (2009). “Using the context-enhanced additive heuristic for temporal and numeric planning”. In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-2009)*.

Fern, Enrique, Erez Karpas, and Brian C Williams (2015). “Mixed Discrete-Continuous Heuristic Generative Planning Based on Flow Tubes”. In: *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI-15)*, pp. 1565–1572.

Fikes, Richard E. and Nils J. Nilsson (1971). “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”. In: *Artificial Intelligence* 2.3-4, pp. 189–208.

Finger, Joseph J. (1987). “Exploiting Constraints in Design Synthesis”. Ph.D. Thesis. Stanford University.

Fox, M, D Long, and K Halsey (2004). “An Investigation into the Expressive Power of PDDL2.1”. In: *Proceedings of the 16th European Conference of Artificial Intelligence (ECAI 2004)*, pp. 328–342.

Fox, Maria and Derek Long (2002). “PDDL+: Modelling Continuous Time-dependent Effects”. In: *3rd International NASA Workshop on Planning and Scheduling for Space*.

Fox, Maria and Derek Long (2003). “PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains”. In: *Journal of Artificial Intelligence Research* 20, pp. 61–124.

Fox, Maria and Derek Long (2006). “Modelling Mixed Discrete-Continuous Domains for Planning”. In: *Journal of Artificial Intelligence Research* 27, pp. 235–297.

Fox, Maria, Alfonso Gerevini, Derek Long, and Ivan Serina (2006). “Plan Stability : Replanning versus Plan Repair”. In: *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS-2006)*, pp. 212–221.

Frank, Jeremy and Paul H. Morris (2007). “Bounding the Resource Availability of Activities with Linear Resource Impact”. In: *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS-2007)*, pp. 136–143.

Garrido, Antonio and Derek Long (2004). “Planning with numeric variables in multiobjective planning”. In: *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI 2004)*.

Gaschler, Andre, Ronald P. A. Petrick, Manuel Giuliani, Markus Rickert, and Alois Knoll (2013). “KVP: A knowledge of volumes approach to robot task planning”. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, pp. 202–208.

Gerbrandy, Jelle and Willem Groeneveld (1997). “Reasoning about Information Change”. In: *Journal of Logic, Language and Information* 6.2, pp. 147–169.

Gerevini, Alfonso and Derek Long (2005). *Plan Constraints and Preferences in PDDL3*. Tech. rep. Department of Electronics for Automation, University of Brescia, Italy, pp. 1–12.

Gerevini, Alfonso and Derek Long (2006). “Preferences and soft constraints in PDDL3”. In: *ICAPS Workshop on Planning with Preferences and Soft Constraints.* Pp. 46–53.

Gerevini, Alfonso and Ivan Serina (2002). “LPG: A Planner Based on Local Search for Planning Graphs with Action Costs.” In: *AIPS 2002*. Vol. 2, pp. 13–22.

Gerevini, Alfonso and Ivan Serina (2003). “Planning as Propositional CSP: From Walksat to Local Search Techniques for Action Graphs”. en. In: *Constraints* 8.4, pp. 389–413.

Gerevini, Alfonso, Ivan Serina, Alessandro Saetti, and Sergio Spinoni (2003a). “Local search techniques for temporal planning in LPG”. In: *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS-2003)*, pp. 62–71.

Gerevini, Alfonso, Alessandro Saetti, and Ivan Serina (2003b). “On Managing Temporal Information for Handling Durative Actions in LPG”. In: *Proceedings of AI*IA 2003: Advances in Artificial Intelligence*. Springer Berlin Heidelberg, pp. 91–104.

Gerevini, Alfonso, Alessandro Saetti, and Ivan Serina (2003c). “Planning Through Stochastic Local Search and Temporal Action Graphs in LPG.” In: *Journal of Artificial Intelligence Research* 20, pp. 239–290.

Gerevini, Alfonso, Alessandro Saetti, Ivan Serina, and Paolo Toninelli (2004). “LPG-TD: a Fully Automated Planner for PDDL2. 2 Domains”. In: *14th International Conference on Automated Planning and Scheduling (ICAPS-2004) IPC Abstracts*.

Gerevini, Alfonso E., Alessandro Saetti, and Ivan Serina (2008). “An Approach to Efficient Planning with Numerical Fluents and Multi-criteria Plan Quality”. In: *Artificial Intelligence* 172.8-9, pp. 899–944.

Gerevini, Alfonso E., Patrik Haslum, Derek Long, Alessandro Saetti, and Yannis Dimopoulos (2009). “Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners”. In: *Artificial Intelligence* 173.5, pp. 619–668.

Gerevini, Alfonso E, Alessandro Saetti, and Ivan Serina (2010). “Temporal Planning with Problems Requiring Concurrency through Action Graphs and Local Search”. In:

Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-2010). AAAI, pp. 226–229.

Ghallab, Malik, Dana Nau, and Paolo Traverso (2004). *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers.

Gregory, Peter, Derek Long, Maria Fox, and J Christopher Beck (2012). "Planning Modulo Theories : Extending the Planning Paradigm". In: *The 22nd International Conference on Automated Planning and Scheduling (ICAPS-2012)*, pp. 65–73.

Hart, Peter E., Nils J. Nilsson, and Bertram Raphael (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107.

Helmert, Malte (2004). "A Planning Heuristic Based on Causal Graph Analysis". In: *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-2004)*. AAAI, pp. 161–170.

Helmert, Malte (2006). "The Fast Downward Planning System". In: *Journal of Artificial Intelligence Research* 26, pp. 191–246.

Helmert, Malte and Carmel Domshlak (2009). "Landmarks, Critical Paths and Abstractions: What's the Difference Anyway?" In: *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS-2009)*. AAAI, pp. 162–169.

Helmert, Malte and Héctor Geffner (2008). "Unifying the Causal Graph and Additive Heuristics". In: *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS-2008)*. AAAI, pp. 140–147.

Helmert, Malte, Patrick Haslum, and Jörg Hoffmann (2007). "Flexible Abstraction Heuristics for Optimal Sequential Planning." In: *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS-2007)*, pp. 176–183.

Hentenryck, Pascal Van and Russell Bent (2006). *Online stochastic combinatorial optimization*. The MIT Press.

Henzinger, Thomas A (2000). "The Theory of Hybrid Automata". In: *Verification of Digital and Hybrid Systems*. Vol. 170. Springer Berlin Heidelberg, pp. 265–292.

Hertle, Andreas, Christian Dornhege, Thomas Keller, and Bernhard Nebel (2012). "Planning with Semantic Attachments : An Object-Oriented View". In: *The 20th European Conference on Artificial Intelligence (ECAI 2012)*. IOS Press, pp. 402–407.

Hildebrand, Francis B. (1987). *Introduction to Numerical Analysis*. 2nd Ed. Dover Publications Inc.

Hoffmann, Jörg (2003). "The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables". In: *Journal of Artificial Intelligence Research* 20, pp. 291–341.

Hoffmann, Jörg (2005). "Where "Ignoring Delete Lists" Works: Local Search Topology in Planning Benchmarks". In: *Journal of Artificial Intelligence Research* 24, pp. 685–758.

Hoffmann, Jörg and Stefan Edelkamp (2005). “The Deterministic Part of IPC-4: An Overview”. In: *Journal of Artificial Intelligence Research (JAIR)* 24, pp. 519–579.

Hoffmann, Jörg and Bernhard Nebel (2001). “The FF Planning System: Fast Plan Generation Through Heuristic Search”. In: *Journal of Artificial Intelligence Research* 14, pp. 253–302.

Howey, Richard, Derek Long, and Maria Fox (2004). “VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning using PDDL”. In: *Proceedings of the 16th IEEE International Conference on Tools with Artificial Intelligence*. IEEE Comput. Soc, pp. 294–301.

Ipakchi, Ali and Farrokh Albuyeh (2009). “Grid of the Future”. In: *IEEE Power and Energy Magazine* 7.2, pp. 52–62.

Iversen, Villy Bæk (2007). “Reversible fair scheduling: the teletraffic theory revisited”. In: *Managing Traffic Performance in Converged Networks*, pp. 1135–1148.

Karmarkar, Narendra (1984). “A new polynomial-time algorithm for linear programming”. In: *Combinatorica* 4.4, pp. 373–395.

Karpas, Erez and Carmel Domshlak (2009). “Cost-optimal planning with landmarks”. In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*, pp. 1728–1733.

Katz, Michael and Carmel Domshlak (2008). “Structural Patterns Heuristics via Fork Decomposition”. In: *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS-2008)*, pp. 182–189.

Kempton, Willett and Steven E. Letendre (1997). “Electric vehicles as a new power source for electric utilities”. In: *Transportation Research Part D: Transport and Environment* 2.3, pp. 157–175.

Keyder, Emil and Héctor Geffner (2008). “Heuristics for Planning with Action Costs Revisited”. In: *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI 2008)*, pp. 588–592.

Khachiyan, L.G. (1980). “Polynomial Algorithms in Linear Programming”. In: *USSR Computational Mathematics and Mathematical Physics* 20.1, pp. 53–72.

Khouadjia, Mostepha, Marc Schoenauer, Vincent Vidal, Johann Dréo, and Pierre Savéant (2013). “Pareto-Based Multiobjective AI Planning”. In: *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-13)*, pp. 2321–2327.

Kim, Yong-Seok (1994). “An optimal scheduling algorithm for preemptable real-time tasks”. In: *Information Processing Letters* 50, pp. 43–48.

Kirkpatrick, S, C D Gelatt, and M P Vecchi (1983). “Optimization by simulated annealing.” In: *Science (New York, N.Y.)* 220.4598, pp. 671–80.

Korf, Richard E. (1985). “Depth-first Iterative-Deepening: An Optimal Admissible Tree Search”. In: *Artificial Intelligence* 27.1, pp. 97–109.

Korf, Richard E., Michael Reid, and Stefan Edelkamp (2001). "Time Complexity of Iterative-Deepening-A*". In: *Artificial Intelligence* 129.1-2, pp. 199–218.

Kovacs, Daniel L (2011). "Complete BNF description of PDDL3.1".

Kowalski, Robert and Marek Sergot (1986). "A Logic-Based Calculus of Events". In: *New Generation Computing* 4.1, pp. 67–95.

Laborie, Philippe (2003). "Resource temporal networks: Definition and complexity". In: *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pp. 948–953.

Landin, Peter John (1964). "The Mechanical Evaluation of Expressions". In: *The Computer Journal* 6.4, pp. 308–320.

Lehoczky, J.P. (1996). "Real-time queueing theory". In: *17th IEEE Real-Time Systems Symposium*, pp. 186–195.

Levesque, Hector J., Raymond Reiter, Yves Lespérance, Fangzhen Lin, and Richard B. Scherl (1997). "GOLOG: A logic programming language for dynamic domains". In: *The Journal of Logic Programming* 31.1-3, pp. 59–83.

Li, Hui and Brian Williams (2011). "Hybrid planning with temporally extended goals for sustainable ocean observing". In: *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-11)*. Vol. 1, pp. 1365–1370.

Li, Hui X and Brian C Williams (2008). "Generative Planning for Hybrid Systems Based on Flow Tubes". In: *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS-2008)*, pp. 206–213.

Lifschitz, Vladimir (1986). "On The Semantics Of STRIPS". In: *Reasoning About Actions and Plans: Proceedings of the 1986 Workshop*. Morgan Kaufmann, pp. 1–9.

Löhr, Johannes, Patrick Eyerich, Stefan Winkler, and Bernhard Nebel (2013). "Domain Predictive Control Under Uncertain Numerical State Information". In: *Proceedings of the 23rd International Conference on Automated Planning and Scheduling*.

Long, D and M Fox (2003a). "The 3rd International Planning Competition: Results and Analysis". In: *Journal of Artificial Intelligence Research* 20, pp. 1–59.

Long, Derek and Maria Fox (2003b). "Exploiting a Graphplan Framework in Temporal Planning". In: *Proceedings of the 13th International Conference on Automated Planning and Scheduling (ICAPS-2003)*. Vol. 1, pp. 52–61.

Luo, Tianyu, Graham Ault, and Stuart Galloway (2010). "Demand Side Management in a Highly Decentralized Energy Future". In: *45th International Universities Power Engineering Conference (UPEC)*.

McCarthy, John (1963). *Programs with Common Sense*. Tech. rep. Defence Technical Information Center, pp. 300–307.

McCarthy, John (1977). “Epistemological Problems of Artificial Intelligence.” In: *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, pp. 1038–1044.

McCarthy, John (1986). “Applications of circumscription to formalizing common-sense knowledge”. In: *Artificial Intelligence* 28.1, pp. 89–116.

McCarthy, John and Patrick J Hayes (1969). “Some Philosophical Problems from the Standpoint of Artificial Intelligence”. In: *Readings in Artificial Intelligence*, pp. 431–450.

McDermott, Drew (1996). “A Heuristic Estimator for Means-Ends Analysis in Planning”. In: *Proceedings of the 3rd International Conference on Artificial Intelligence Planning Systems (AIPS-96)*, pp. 142 –149.

McDermott, Drew, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins (1998). *PDDL - The Planning Domain Definition Language*. Tech. rep. Yale Center for Computational Vision and Control.

McIlraith, Sheila and Ronald Fadel (2002). “Planning with Complex Actions”. In: *Proceedings of the 9th International Workshop on Non-Monotonic Reasoning (NMR 2002)*, pp. 356–364.

McIlraith, Sheila A (1997). “Representing Actions and State Constraints in Model-Based Diagnosis”. In: *Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI-97 / IAAI-97)*, pp. 43–49.

McIlraith, Sheila A. (2000). “Integrating Actions and State Constraints: A Closed-Form Solution to the Ramification Problem (Sometimes)”. In: *Artificial Intelligence* 116.1-2, pp. 87–121.

Mirkis, Vitaly and Carmel Domshlak (2007). “Cost-Sharing Approximations for h+”. In: *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS-2007)*. AAAI Press, pp. 240–247.

Mitchell, John C. and Gordon D. Plotkin (1988). “Abstract types have existential type”. In: *ACM Transactions on Programming Languages and Systems* 10.3, pp. 470–502.

Moore, JM (1968). “An n job, one machine sequencing algorithm for minimizing the number of late jobs”. In: *Management Science* 15.1, pp. 102–109.

Nakhost, Hootan and Martin Müller (2009). “Monte-Carlo Exploration for Deterministic Planning”. In: *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI-09)*.

Nakhost, Hootan and Martin Müller (2010). “Action Elimination and Plan Neighborhood Graph Search : Two Algorithms for Plan Improvement”. In: *Proceedings of the 20th International Conference on Automated Planning and Scheduling (ICAPS-2010)*, pp. 121–128.

Nakhost, Hootan and Martin Müller (2013). “Towards a second generation random walk planner: an experimental exploration”. In: *Proceedings of the 23rd International Joint Conference of Artificial Intelligence (IJCAI-13)*, pp. 2336–2342.

Nieuwenhuis, Robert, Albert Oliveras, and Cesare Tinelli (2006). “Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T)”. In: *Journal of the ACM (JACM)* 53.6, pp. 937–977.

Ortega, J. M. and W. C. Rheinboldt (1970). *Iterative Solution of Nonlinear Equations in Several Variables*. Vol. 30. Society for Industrial and Applied Mathematics.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. English. Addison-Wesley.

Pednault, Edwin P D (1989). “ADL: Exploring the Middle Ground Between STRIPS and the Situation Calculus”. In: *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 324–332.

Pednault, Edwin P D (1994). “ADL and the State-Transition Model of Action”. In: *Journal of Logic and Computation* 4.5, pp. 467–512.

Penberthy, J. Scott and Daniel S. Weld (1994). “Temporal Planning with Continuous Change”. In: *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pp. 1010–1015.

Petrick, Ronald P. A. and Andre Gaschler (2014). “Extending Knowledge-Level Contingent Planning for Robot Task Planning”. In: *Proceedings of the ICAPS 2014 Workshop on Planning and Robotics (PlanRob)*, pp. 157–165.

Piacentini, Chiara, Varvara Alimisis, Maria Fox, and Derek Long (2015). “An Extension of Metric Temporal Planning with Application to AC Voltage Control”. In: *Artificial Intelligence* 229, pp. 210–245.

Pina, André, Carlos Silva, and Paulo Ferrão (2012). “The impact of demand side management strategies in the penetration of renewable electricity”. In: *Energy* 41.1, pp. 128–137.

Pohl, Ira (1970). “Heuristic search viewed as path finding in a graph”. In: *Artificial Intelligence* 1.3-4, pp. 193–204.

Quemy, Alexandre and Marc Schoenauer (2015). “True Pareto Fronts for Multi-Objective AI Planning Instances”. In: *In Proceedings of the 15th European Conference on Evolutionary Computation in Combinatorial Optimisation (EvoCOP 2015)*. Vol. 9026. Springer Verlag, pp. 197–208.

Rawlings, J.B. (2000). “Tutorial overview of model predictive control”. In: *IEEE Control Systems Magazine* 20.3, pp. 38–52.

Reiter, Raymod (2001). *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Reiter, Raymond (1991). “The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression”. In: *Artificial intelligence and mathematical theory of computation: papers in honor of John McCarthy* 27, pp. 359–380.

Richter, Silvia and Matthias Westphal (2010). “The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks”. In: *Journal of Artificial Intelligence Research* 39, pp. 127–177.

Richter, Silvia, Malte Helmert, and Matthias Westphal (2008). “Landmarks Revisited.” In: *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pp. 975–982.

Rintanen, J (2007). “Complexity of Concurrent Temporal Planning”. In: *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS-2007)*, pp. 280–287.

Rowe, M, W Holderbaum, and B Potter (2013). “Control Methodologies : Peak Reduction Algorithms For DNO Owned Storage Devices On The Low Voltage Network”. In: *4th IEEE PES Innovative Smart Grid Technologies Europe (ISGT Europe)*, pp. 1–5.

Russell, Stuart Jonathan and Peter Norvig (2009). *Artificial Intelligence: A Modern Approach*. 3rd ed. Prentice Hall.

Scott, Paul, Sylvie Thiébaux, and Pascal Van Hentenryck (2013). “Residential Demand Response under Uncertainty”. In: *19th International Conference on Principles and Practice of Constraint Programming (CP-13)*, pp. 645–660.

Shanahan, Murray (1990). “Representing Continuous Change in The Event Calculus”. In: *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI-90)*. Pitman, pp. 598–603.

Shin, Ji-Ae and Ernest Davis (2005). “Processes and Continuous Change in a SAT-Based Planner”. In: *Artificial Intelligence* 166.1-2, pp. 194–253.

Smith, David E and Daniel S Weld (1999). “Temporal Planning with Mutual Exclusion Reasoning”. In: *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pp. 326–337.

Sroka, Michal and Derek Long (2012). “Exploring Metric Sensitivity of Planners for Generation of Pareto Frontiers”. In: *In Proceedings of the Sixth Starting AI Researchers’ Symposium (STAIRS 2012)*, pp. 306–317.

Strbac, Goran (2008). “Demand side management: Benefits and challenges”. In: *Energy Policy* 36.12, pp. 4419–4426.

To, Son Thanh, Mark Roberts, Thomas Apker, Benjamin Johnson, and David W. Aha (2016). “Mixed Propositional Metric Temporal Logic : A New Formalism for Temporal Planning”. In: *Workshops of the 30th AAAI Conference on Artificial Intelligence. Planning for Hybrid Systems: Technical Report WS-16-12*, pp. 631–640.

Trinquart, Romain and Malik Ghallab (2001). “An Extended Functional Representation in Temporal Planning: Towards Continuous Change”. In: *Proceedings of the European Conference on Planning (ECP)*, pp. 203–208.

Van Ditmarsch, Hans, Wiebe Van der Hoek, and Barteld Kooi (2007). *Dynamic Epistemic Logic*. Vol. 337. Springer Science & Business Media.

Vidal, Vincent (2004). "A lookahead strategy for heuristic search planning". In: *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-2004)*, pp. 150–159.

Wang, David and Brian C. Williams (2015). "tBurton: A Divide and Conquer Temporal Planner". In: *Proceedings of the 29th AAAI Conference on Artificial Intelligence (AAAI-15)*, pp. 3409–3417.

Weld, Daniel S. (1994). *An Introduction to Least Commitment Planning*.

Xie, Fan, Hootan Nakhost, and Martin Müller (2012). "Planning Via Random Walk-Driven Local Search". In: *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS-2012)*, pp. 315–322.

Yixing Xu, Le Xie, and C. Singh (2010). "Optimal scheduling and operation of load aggregator with electric energy storage in power markets". In: *North American Power Symposium 2010*. IEEE, pp. 1–7.

Yu, Zhe, Linda Mclaughlin, Liyan Jia, Mary C Murphy-hoye, Annabelle Pratt, and Lang Tong (2012). "Modeling and Stochastic Control for Home Energy Management". In: *Power and Energy Society General Meeting 2012*. IEEE, pp. 1–9.

Zehir, M. Alparslan and Mustafa Bagriyanik (2012). "Demand Side Management by controlling refrigerators and its effects on consumers". In: *Energy Conversion and Management* 64, pp. 238–244.